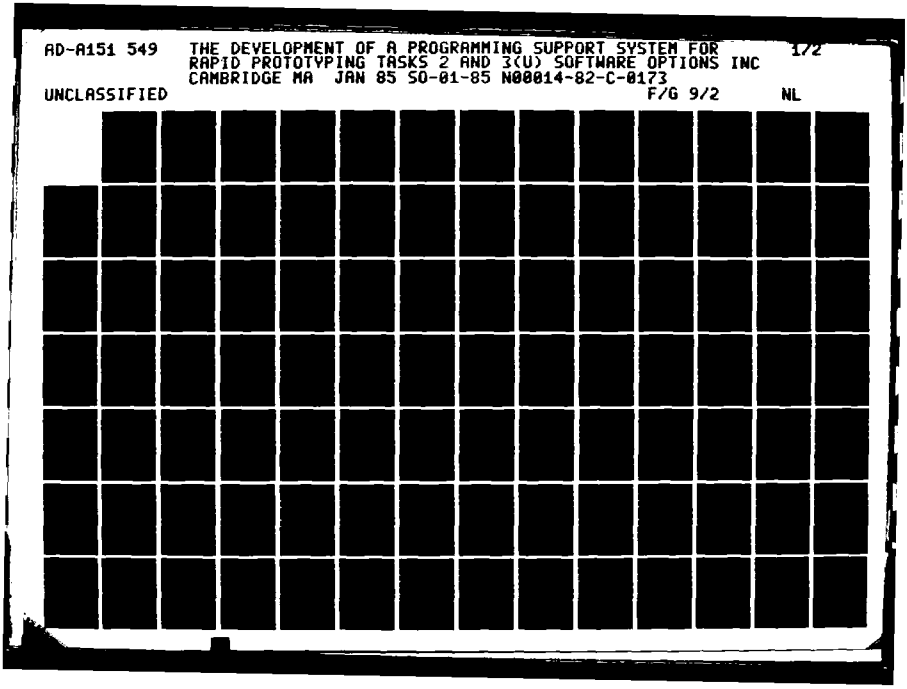
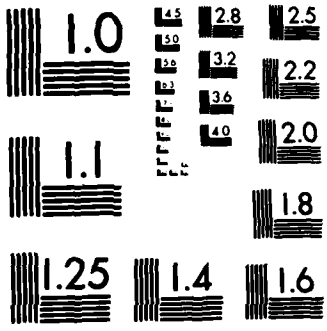

AD-A151 549 THE DEVELOPMENT OF A PROGRAMMING SUPPORT SYSTEM FOR 1/2
RAPID PROTOTYPING TASKS 2 AND 3(U) SOFTWARE OPTIONS INC
CAMBRIDGE MA JAN 85 SO-01-85 N00014-82-C-0173
UNCLASSIFIED F/G 9/2 NL



FINAL REPORT
The Development of a Programming Support System for Rapid Prototyping

N00014 - 82 - C - 0173
January 1984 - January 1985
SO-01-85

Prepared for
Office of Naval Research
Department of the Navy
800 N. Quincy Street
Arlington, Virginia 22217

By

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138
Tel. (617) 497-5054

Summary

This report describes the results of work on Tasks 2 and 3 of this project, initially conceived to be a five year project to develop a programming support environment and a collection of tools that support rapid prototyping. For a broader overview of the project, in particular, of the support environment, see the final report for Task 1 [1]. The initial project was scaled back, and we report here on narrow aspects of the problems that played a role in the larger system.

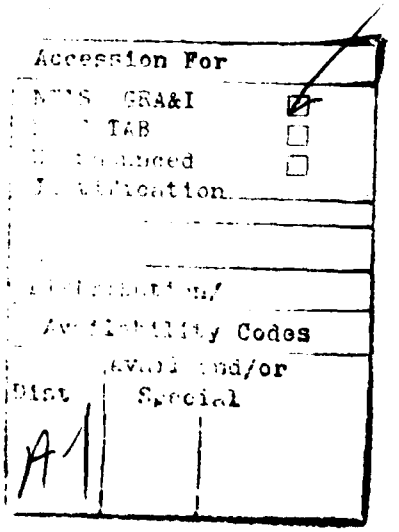
The principal work in Task 2 was the design of a new method for code-generation, particularly oriented to the needs and capabilities of the programming support environment. This resulted in a rather large, self-contained document, Code Generation by Coagulation, which is included in its original form as part of this report.

Task 3 was to have been an effort to prototype some of the code-generation ideas developed in Task 2, in particular, an analyzer that builds an intermediate form for bi-directional scanning of a program, a necessary constituent of the optimizing code-generator. Task 3 also called for developing overall specifications for the Rulog language and interpreter and developing a prototype of the interpreter. Due to the limitation of funds, only a design, not a prototype, of the bi-directional

scanner was eventually supported; this work is reported on in the second document included in this report. The work on Rulog is reported on in a paper that has been submitted to the 8th International Conference on Software Engineering; a copy of that paper is attached.

Bibliography

1. "The Development of a Programming Support System for Rapid Prototyping - Final Report for Task 1", Technical Report SO-01-83, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138.



Code Generation by Coagulation

14 December 1983

Michael Karr

Software Options, Inc.
22 Hilliard Street
Cambridge, MA 02138

This paper describes a new approach to code-generation. The central tenet is that there must be a more intimate coupling between register allocation and instruction selection than exists in present-day technology. This is achieved by generating code in very small regions and gradually coalescing the part of the program that is "compiled".

This work was supported in part by ONR contract N00014-82-C-0173.

1. Introduction

1.1 Traditional Assumptions

The problem of designing a code generator for an interpreter-based language has provided a chance to re-examine the traditional assumptions underlying the present technology of code-generation. Probably the most pervasive of these assumptions is that programs are compiled before they are run. In an interpretive environment, a compiler sees a program only after it is mostly debugged; changes to a program are usually tested with the interpreter before the program is recompiled. This comparative infrequency of compilation has two consequences, one of which is obvious: a compiler for such an environment can run slower than other compilers, since it is used less often. The less obvious implication of having run programs before they are compiled is that one can design a compiler whose optimization techniques are fundamentally dependent upon execution statistics gathered when running the program on "typical" data.

Another assumption underlying present compilers is that subroutines are generally large and are called infrequently. While this was less absurd twenty years ago, language implementations continue to have rigid calling sequences, register conventions, etc. As programmers are being taught more and more to break down their programs into smaller, more easily understood pieces, there is a growing importance of not punishing them with increased computing costs. A variety of parameter and result passing mechanisms, convenient register conventions between caller and callee, cheap subroutine calls to non-recursive procedures (not to mention direct substitution when possible) are all required to support good inter-procedural optimization at the machine code level.

Code space optimization is an area neglected in most compilers, as if space were free. A corollary to this is that time-space trace-offs are also neglected, particularly because execution data is lacking. Though some compilers do indeed try to pay special attention to "inner loops", the notion of inner loop becomes tenuous when considering inter-procedural optimization. With the variety of computing environments proliferating, concerns with space costs increase. On small personal computers and in some embedded systems, the space cost may be of paramount

concern; in time-sharing environments, cost is usually a function of space and time, so trade-offs are crucial in the overall optimization process; in paged systems, space can convert indirectly into time because of page faults; in real-time systems, time costs may dominate.

1.2 Traditional Techniques

In the initial stages of this code-generator design, there were two issues which stood out as unsuccessfully treated in compiler designs with which I was familiar. First, there was the fact that while code-generation is an optimization problem, the objective function (to use a term from mathematical programming) does not enter in a direct way into the process; rather it forms an implicit background for all that happens. The reason for this is partly that execution information has been assumed to be unavailable to compilers (see above); one of the consequences is that time-space trade-off questions have been neglected (again, see above). The second bothersome issue is best summed up in the standard cliche about code-generators: instruction selection is trivial once register allocation is done, and register allocation is trivial once instruction selection is done.

Consider peephole optimization. This is one of the last phases of a compiler. Its job is to rummage about in the code which has already been generated, to remove obvious inefficiencies, and to detect patterns which can be more efficiently compiled using instructions which were not issued by earlier phases in the compiler. It is necessary to realize the important role played by this technique in today's best optimizing compilers [8]: "It is without doubt the most ad hoc, least formalized, and perhaps least aesthetically pleasing of the phases of the compiler. Yet it is one of the most effective". Even if peephole optimization is effective, it is disturbing that it works so well—peephole optimizations are often allowed by the fact that certain quantities are in registers (instruction selection is trivial if ...), but register allocation was done much earlier, and might have been done differently, had there been any knowledge of the effect it would have on peephole optimization (register allocation is trivial if ...). And all of this is happening without any clear view of the ultimate effect on performance.

1.3 A New Approach

With a new set of assumptions in mind, and with reservations about some aspects of present code-generation techniques aroused, we begin to describe our proposed design. The place to begin is with the circularity cliche, and the driving idea is never to break the circularity—instruction selection and register allocation *always* are done together. This may seem impossible, but the trick is to look at a small enough piece of the program, so that doing both is not only possible, but easy. Instead of beginning by code generation (both register allocation and instruction selection) which is *safe globally*, and patching it up with a peephole optimizer, we propose to generate code which is *optimal locally*, and gradually paste the pieces together into a coherent whole, modifying both register allocations and instructions in the process.

The order in which pieces are pasted together is crucial to this approach. It is done in order of decreasing execution frequency, the idea being to get things properly arranged on the expensive paths through the program. For example, inner loops which are truly busy will be compiled first, registers arranged, etc. But those pieces inside loops that are seldom used will have no influence on the initial register assignments. When compiling these pieces, if there is some new register problem, there are several ways to resolve it. Either the present code can be changed to make it compatible, or the new piece can be compiled to work around existing conventions. The relative costs of the two methods can be compared, and a rational choice made.

By now, the meaning of "coagulation" in the title of this work should be clearer. Imagine the program, including the subroutines, spread out over a table, with the compiler dropping Jello on the parts as they are compiled. At first little drops appear in seemingly random places. These get bigger and combine with other drops to form growing globs. When two globs meet, ripples will go out through each as they adjust to each other's presence, although the parts of the globs that formed first are less affected by the ripples. When compilation is complete, there is one congealed mass.

1.4 Scope and Limitations

This work is about code-generation, not about the entire process of compilation. We will assume that before a compiler begins generating code, it will already have extensively analyzed the program, and produced an intermediate form. While this is

not the place to discuss intermediate form (see chapter 5), we will claim here that conventional languages can all be reduced to quite similar intermediate forms. There may be great variations in surface syntax and applications among COBOL, LISP, FORTRAN and ADA, but from the point of view of code-generation there is little difference. On the other hand, this work makes no attempt to treat highly unconventional languages. For example, the issues in a PROLOG compiler are simply not considered here. Neither do we consider query languages for relational databases.

Any discussion of code-generation must consider target architecture. The techniques presented here apply to conventional machines, such as the PDP-10, IBM-370, MC68000, and VAX. These are characterized by generally serial operation, on the order of 10 registers, and instructions that usually operate on or via the registers. Special purpose machines, for example, the SCHEME chip, probably would not benefit from this new approach; nor would highly pipelined or vector machines (we certainly do not address the problem of parallelizing serial programs). Although there is no intent to address the issue specifically, this design may be useful in the generation of micro-code, where the problem is to have the data in the right place at the right time.

1.5 A Guide for the Reader

This work develops the techniques that are necessary to make the coagulation idea into an algorithm—albeit a large one—for code-generation. There is much further work that could be done; indications of topics to explore further are described as they arise in the course of the discussion. The most pressing issue is that of an implementation, which has not yet started. It is too often necessary to apeal to one's sense of what is likely to be found in real programs, rather than referring to evidence gathered by a compiler in everyday use.

For the reader wishing to skim this work, there is unfortunately a rather sequential dependence of chapters. The best approach is to read from the beginning, until tired. Chapter 2 outlines the mathematical objects that we will study. A natural stopping point for the casual reader is at the end of this chapter. Chapter 3 considers in more detail two relations introduced in Chapter 2, cohabitation and conflict. These relations capture the competing influences in code-generation—the desire to

have values remain in the same place, for speed, and the necessity to have values be in different places, to preserve the meaning of the program. The reader will have a much better idea of coagulation at the end of Chapter 3.

Chapter 4 presents a technical device that is useful in representing the cohabitation and conflict relations, and in detecting inconsistency between the two. Skipping this chapter on first reading will cause only momentary confusion in Chapter 5, but will leave the reader unprepared for Chapter 6. Chapter 5 provides an even better perspective on coagulation, because of the thoroughgoing way in which it follows the imperative to be optimal locally, rather than safe globally.

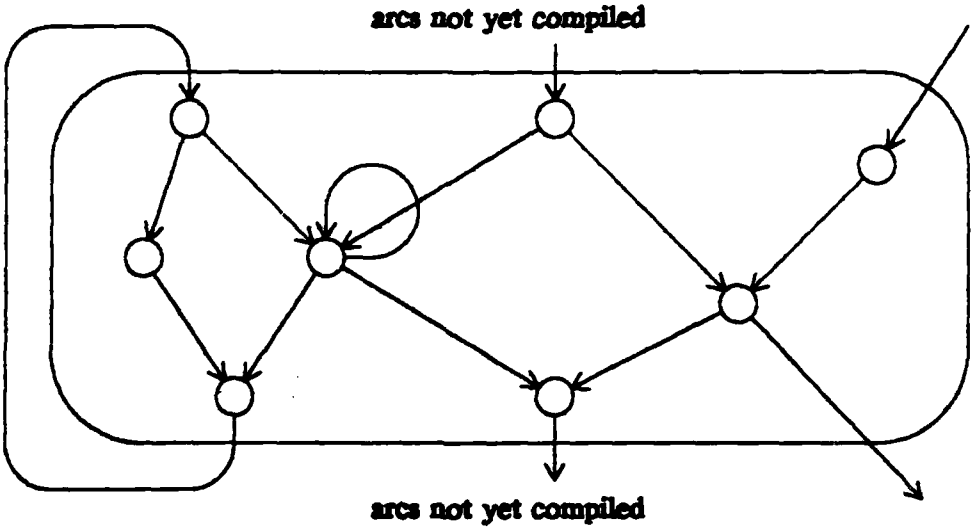
Real coagulation—techniques for joining previously "compiled" but unrelated pieces—is the subject of the rest of this work. Chapter 6 provides algorithmic details for enlarging cohabitation and conflict relations, and for determining whether the new relations are still consistent. Chapter 7 lays the groundwork for dealing with inconsistencies, and shows that inconsistencies have one of two distinct forms, splits and twists. Chapters 8 and 9 give techniques to deal with the two kinds of inconsistencies.

2. A Glimpse of the Basic Concepts

In this chapter we give a brief preview of various ideas used in building the code generator. With these in view, the more detailed descriptions of both processes and data structures will be better motivated.

2.1 Regions

We view the program as being represented by a traditional flowgraph. A *region* is a subgraph of this flowgraph (possibly consisting of a single node) which has already been compiled.

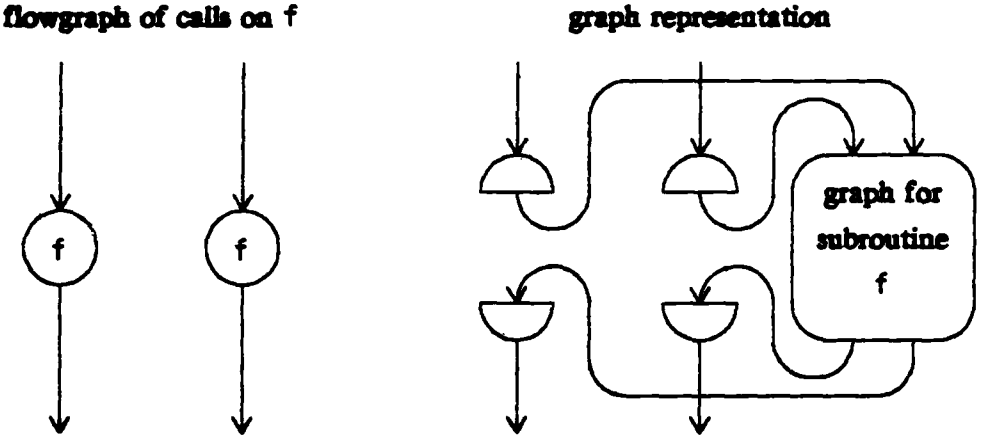


We speak of the compilation of nodes and arcs separately. Compilation of a node produces a new region, consisting solely of the node. This compilation yields a sequence of machine instructions for the part of the program which lies in the node, as well as other data associated with the region. Thus, a node is not necessarily a maximal flowblock, but rather the largest piece of program which can conveniently be turned into a region. This might be no larger than a single operation, e.g. T <- V + W.

The compilation of an arc is the interesting part. This happens only when the nodes at each end of the arc have been compiled, i.e., each node is in some region—perhaps much larger than a single node. After an arc has been compiled it means roughly that the code prior to this arc is compatible with the code following the arc: for example, registers are compatibly assigned. If an arc is compiled both of whose

nodes already lie in the same region, it merely has the effect of adding an arc to a region. This is called *intra-region* compilation. If the arc connects heretofore distinct regions, then after it is compiled, there is only one region, consisting of the two smaller ones, plus the added arc. This is referred to as *inter-region* compilation.

We note here that calls on a routine which is being compiled are represented (at least conceptually) by graph structure.



When an arc entering a subroutine graph is compiled, it means that the argument conventions (as well as register conventions, etc.) are mutually understood by caller and callee; similarly with an arc leaving a subroutine graph and result conventions.

The execution data required for the compilation process is frequency counts on the arcs (not merely on the nodes). Throughout this document, the *frequency* of an arc will mean the number of times flow passes through the arc during one execution of the program. The top-level description of the compiler may be summarized as follows:

Algorithm 2.1 Compilation

for $A \leftarrow$ each arc in order of decreasing frequency
    if the entry node to A is uncompiled **then** compile it
    if the exit node to A is uncompiled **then** compile it
    Compile the arc A.

2.2 Cohabitation and Conflict

Recall that the nodes in a region have compiled code associated with them. A line of code consists of an opcode followed by zero or more operands. Whenever a variable is one of these operands, it is referred to as an *occurrence* (i.e., of the variable). Thus, a line of code might be ADD V,W. To refer to an occurrence of a particular variable, we use the notation V_n where n is often 1. Beware that V_1 and V_2 are not different variables, but different occurrences of the same variable. Thus, the above line of code would usually be written ADD V_1,W_1, so that we could refer to the occurrences V_1 and W_1. If we wish to talk about an occurrence without naming its variable, we use o, possibly subscripted. The variable of an occurrence is denoted by v, so "o_1 and o_2 have the same variable" is written $v(o_1) = v(o_2)$.

There are two important relations on occurrences. If two occurrences *cohabit*, it means that the present code relies upon the fact that the two occurrences are in the same memory location ("memory" here includes register and stack locations). Thus cohabitation is an equivalence relation, partitioning occurrences into *cohabitation classes*. Cohabitation is the *only* way in which the same memory location is referred to by different instructions. The code corresponding to different (in the source code) occurrences of the same variable may find "the variable" in quite different places. Moreover, occurrences of different variables may cohabit, if that is a useful optimization.

The second relation of interest is that of *conflict*. If two occurrences conflict, it means that present code relies upon the fact that the two occurrences do *not* occupy the same memory location. There is a simple rule relating cohabitation and conflict:

> Two occurrences which are in conflict may not be a member of the same cohabitation class.

This is called the consistency rule; much more will be said about it, or rather, about inconsistency.

2.3 Supply and Demand Sets

Define an *entry node* of a region to be any node in that region which has an uncompiled incoming arc; dually, an *exit node* is one which has an uncompiled outgoing arc. A *boundary node* of a region is any entry node or exit node of the region.

Following standard terminology, we call a variable *live* at some point in the flowgraph if there is some execution path leaving that point along which the variable is used before it is set. Each entry node for a region specifies the set of variables which are live at entry to the node, and which can be seen to be live solely by looking at the region. This set of variables is represented by a set of occurrences, called the *demand set*. Similarly, each exit node for a region specifies the set of variables which are live at exit from the node and which are mentioned in the region. As before, this set of variables is represented by a set of occurrences, the *supply set*. Either of these sets may be referred to as a *boundary set*. The calculation of boundary sets presupposes a pass over the program to do live-dead analysis. Further, the implications of the phrase "mentioned in the region" are more subtle than one might expect. The details of both these issues are discussed in chapter 4.

The purpose of the boundary sets is to aid in the compilation of an arc. A variable may have occurrences in both the supply and demand sets, in which case the occurrences must be made to cohabit. It is also possible for a variable to have an occurrence in the boundary set of only one of the regions. Only variables that are both live and mentioned in a region are necessary in the coagulation process, and only those are included in boundary sets.

2.4 The Cost Metric

The objective function for the optimization process is the cost of running the program on the same data which generated the arc frequencies that govern order of compilation. We assume this metric to be a bilinear function of average space S and total time T:

$$\text{cost}(S, T) = a_1 \cdot S \cdot T + a_2 \cdot T + a_3 \cdot S + a_4$$

This formula covers most charging policies, and the coefficients have reasonable interpretations:

- a_1 is the cost per unit space per unit time
- a_2 is the cost per unit time of the CPU
- a_3 is a job surcharge for space
- a_4 is a job submission charge

In making decisions, we are interested in incremental cost:

$$\text{cost}(S + s, T + t) - \text{cost}(S, T) = a_1 \cdot (S \cdot t + T \cdot s + s \cdot t) + a_2 \cdot t + a_3 \cdot s$$

The term $s \cdot t$ will generally be much smaller than the other terms, and may be

ignored. Rearranging the rest, we have:

$$(a_1 \cdot T + a_3) \cdot s + (a_1 \cdot S + a_2) \cdot t$$

Whenever we are changing or adding an instruction, it is simple to determine the extra space involved (s), and since we know the frequency of the instruction, we know the extra time involved (t).

The missing important parameters are the overall space S and overall time T. Since the program has been run, there should be a known value for S. If it was run in an interpretive environment, there will be some change in space due to compilation; since at least one pass has been made over the program before code-generation begins, the size of the source will be known. A little experience with the compiler should give a reliable conversion factor from source size to object code size, so S can be estimated. Of course, if the program was run in its compiled form, we have actual experience (probably space doesn't change too much with small changes in the program). It must be remembered that most of S may be space for data, not for program, so that errors in the estimated size of the program do not gravely affect the overall estimate.

The estimation of T is trickier, but the same ideas apply. Since we can assume that the source has frequency statements attached, a little experience should give a usable estimate for the overall time of the compiled program. Direct experience with previous compilations of the program being compiled will of course be more reliable. But here we *do not* have a cushion analogous to the one that exists for space estimation.

Once there are estimates for S and T, we can obtain the basic time-space trade-off factor:

$$b = (a_1 \cdot S + a_2)/(a_1 \cdot T + a_3)$$

Thus, when changing the program in a way which uses s extra units of space and is executed f times, the extra cost is proportional to:

$s + c \cdot f$ where $c = b \cdot$ time to execute the instruction

The expression $s + c \cdot f$ will appear often in this paper, as the generic cost of adding or modifying an instruction. Realize that s and c are determined by the particular instruction.

Using this analysis, we can gain some quantitative insight into the time-space

problem. For simplicity, assume $a_2 = a_3 = 0$ (or are negligible). Then $b = S/T$, so a_1 is irrelevant to studying trade-off. Suppose that we have a 10K program, with 40K of data, so S = 50K and that the program runs in T = 100 seconds; thus b = 500 (with units of words/sec). Suppose we are trying to decide between using 2 instructions which take 2 μs each versus one instruction that takes 5 μs (for all instructions, s = 1). Then, the two instruction sequence is preferable under the condition:

$$(2 \text{ inst}) \cdot (1 \text{ word/inst} + (500 \text{ words/sec}) \cdot (2 \cdot 10^{-6} \text{ sec/inst}) \cdot f) < (1 \text{ inst}) \cdot (1 \text{ word/inst} + (500 \text{ words/sec}) \cdot (5 \cdot 10^{-6} \text{ sec/inst}) \cdot f)$$

$$\Leftrightarrow f > 2000$$

The question is, how often do "typical" instructions get executed in 100 secs? Programs are quite uneven in the distribution of their runtime. Assume that 90% of the runtime accrues uniformly in 10% of the code, and 10% of the runtime accrues uniformly in the other 90% of the code. Estimate the average instruction time as 2 μs, and compute the frequencies of the busy and non-busy parts of the program in a 100 second execution.

busy frequency = 45000 non-busy frequency = 555

Thus, we should use the two instructions in the busy part of the program, but only the one in the non-busy part. One cannot help but one wonder how many programmers would exercise the correct judgment intuitively, and whether it would be worth their time to do the calculation every time a question came up.

While this example shows that time-space trade-offs can arise in the selection of instructions, the importance of such analysis will probably lie in deciding upon other optimization strategies. For example, back-substitution of subroutines can be quite expensive in terms of space. It can also produce dramatic time savings, particularly when applied to small data-structure access routines which the programmer may have defined for the sake of modularity. Using the analysis we have outlined above, it is possible to decide which back-substitutions really do pay off. Another optimization technique involving time-space trade-offs is that of loop unrolling. Only when the cost of space is accounted for does one know when to stop the unrolling process.

2.5 Register Allocation Representation

Associated with each region is a structure representing required and possible register allocation. The "required" attribute means that this structure records which of the cohabitation classes are assumed to be kept in which registers, while the "possible"

attribute means that a definite allocation is not given, only that from the structure it is easy to assign registers in such a way that the presently generated code will work (thereby providing an existence proof that allocation is feasible). For example, if a machine has n identical registers, this data structure can simply be a set, of size not exceeding n, of sets of cohabitation classes. Each of the sets of cohabitation classes would be assigned to the same register. In this scheme, there is clearly a requirement that any two members of the same set of cohabitation classes not be in conflict; any set (of sets) of size less than or equal to n and obeying this requirement would specify a correct register allocation.

3. More on Cohabitation and Conflict

In this chapter we lay out the general approach to the building and maintaining of the cohabitation and conflict relations. Precise details will be given later; the idea here is to show what kind of thinking motivates the details. We conclude with an example showing an optimization which arises naturally in this code generator.

3.1 Calculation

We now give a rough description of when cohabitation and conflict are established. Cohabitation arises in two different ways. The most obvious is the flow of control from one occurrence of a variable to the next occurrence of the same variable. In the compiled code for a region, if flow of control can pass from one occurrence of a variable to another occurrence of the variable without passing over an intervening use of the variable, then the two occurrences must reference the same location. As we stated earlier, the only way for this to happen is for the occurrences to cohabit.

A less obvious way for cohabitation to arise is from the assignment of scalar variables. This is in part a consequence of the dictum that code is generated in the most efficient way for the smallest possible context. When confronted with a statement of the form V <- W, the compiler takes the optimistic approach that nothing at all has to be done here, because it can be arranged for V and W to occupy the same location. This may seem insanely optimistic, but it is done not so much because the programmer may have inserted needless assignments, but because of internal reasons.

> Parameter passing is modeled as assignment; the cohabitation of actual and formal parameter means that the argument to the subroutine is left in exactly the right place. To encourage this, the initial assumption is that it is possible.
>
> Since trivial assignments are optimized away, earlier phases in the compiler need not worry about creating extra assignments, if that is a convenient way to express a transformation.
>
> The programmer's assignments may be necessary, but they may be in the wrong place, from an optimization point of view. By assuming control over them, it is easier for the compiler to produce better code.

In summary, the minimization of moving things around is one of the central problems of low-level optimization. The compiler takes complete control of this, not

allowing itself to be influenced by the programmer's assignments. Thus the *only* way that move instructions are generated is when all of the optimistic assumptions lead to trouble, i.e., to inconsistency. Not surprisingly, inconsistencies can always be resolved with move instructions; the problem is to do so as efficiently as possible, and especially, to avoid moves when possible.

To discuss conflict, we must first discuss generations. A *generation* is an occurrence which is modified by an instruction. (We will call the left hand side of an assignment statement a *first use*.) We will denote generations with an asterisk superscript, as in V_1^*. This convention will also help in reading the generic machine language used in the examples—the asterisked occurrence is the destination of the instruction.

Conflict arcs are established when one occurrence is "propagated past" an occurrence which is a generation. For example, consider the code sequence:

```
MOVE   V1*,W1
ADD    X1*,Y1
```

The occurrences X_1^* and Y_1 must both conflict with V_1^*, because V is being changed, and by the semantics of the language this is not supposed to affect X and Y (assume no sharing here). It is sufficient to establish conflict only when propagating past generations, as we shall see in section 6.1.

3.2 Representation

We have seen that cohabitation is an equivalence relation. We have also seen that cohabitation arises locally, because of flow or assignment. For inconsistency resolution, it is useful to keep track of the individual "reasons" for the existence of a cohabitation class. We do this with a *cohabitation graph*, whose nodes are occurrences and whose arcs arise from the flow of data from one occurrence to the next, or from assignments. It is convenient to let this be a directed graph, with arrows in the direction of the flow of data. A *cohabitation class* corresponds to a connected component of the cohabitation graph.

Since conflict is a relation on occurrences, it too may be viewed as a graph whose nodes are occurrences. This graph and the cohabitation graph share node sets, so it is often convenient to draw them on the same set of nodes, and distinguish arc types.

The pictures are:

These relations are not static during the course of a compilation, but are continually adjusted as regions grow and coalesce. The cohabitation relation will be represented in part by a graph. Because the conflict relation is very dense, its representation and manipulation as a graph would be very expensive; instead, it is represented indirectly, by means explained in chapter 6.

3.3 Costs on Cohabitation Arcs

When an inconsistency arises, it must be resolved. This is done on the basis of costs assigned to each cohabitation arc, which we think of as the cost of "breaking" the cohabitation. The problem of deciding what number to assign as the cost of a cohabitation arc is more difficult than simply deciding whether to establish the arc. This difficulty arises because to assign a number, it is necessary to anticipate how an arc might be broken. We consider the details of this problem later, and focus here on the general principles used.

The breaking of a cohabitation arc usually involves adding some instruction(s), or using more expensive variants of already generated instructions. If we knew what these instructions were, we could use the cost metric described earlier to obtain the appropriate cost for the arc. The problem is that at the time an arc is being established, it is not worthwhile to *determine* these instructions. This is partly because there is no reason to spend a lot of time trying to figure out how to break arcs whose breaking will never be helpful in resolution, and partly because breaking an arc takes place in the context of inconsistency resolution, so that the best way to do it depends upon a larger context. Rather than try to obtain the cost exactly, what is done is to get a good *lower* bound on the cost of breaking the arc, i.e., we *under*-estimate the cost of resolution, but by as little as possible. When the time comes to resolve an inconsistency, the approach is as follows:

1. A set of arcs to be broken is chosen on the basis of the costs on the arcs.
2. Given the set of arcs in step 1, the precise modifications are determined and the precise cost of breaking this set is calculated. If this turns out to be much more than expected, the modifications are remembered, but

step 1 and this step are repeated, looking for a better set to break.

3. Eventually, the step 1-2 loop stops, and we pick the set with minimum actual cost.

If, in step 2, we discover that an arc is more expensive to break than was originally anticipated, the cost of the arc may be revised, so that future calculations in step 1 will have a more accurate view of things. This rise in costs is one of the ways in which the steps 1-2 loop terminates – eventually, the actual cost is close to the estimated, and the estimated cost is known to be about as good as possible. So we use the modifications which have been calculated.

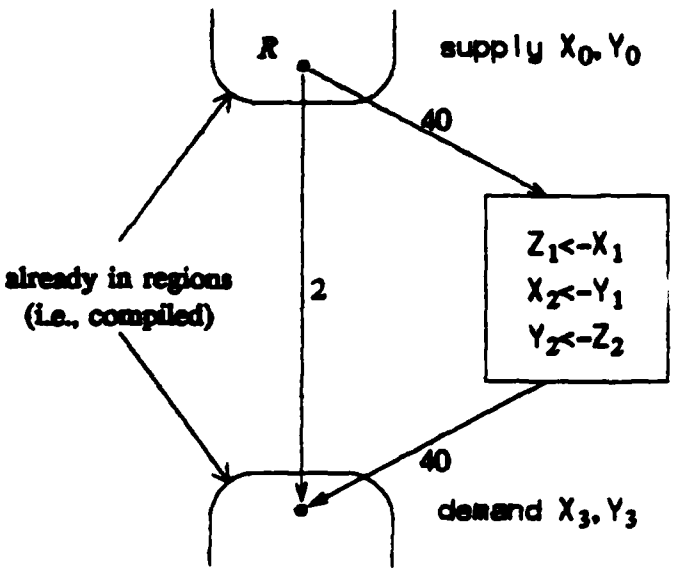
3.4 An Example

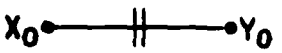
At this point, we offer an example which shows how cohabitation, conflict, and inconsistency resolution interact when compiling a program. The reader will have to accept some statements on faith, such as costs on arcs, choice of arcs to break, and how the compiler chooses to implement the breaks.

The example we choose is a conditional exchange, i.e., a statement such as:

```
IF ... THEN Z <- X; X <- Y; Y <- Z ENDIF ...
```

By the time the code generator sees this, we may assume we are dealing with the flowgraph fragment and conflict relation (numbers near the arcs denote their frequency):



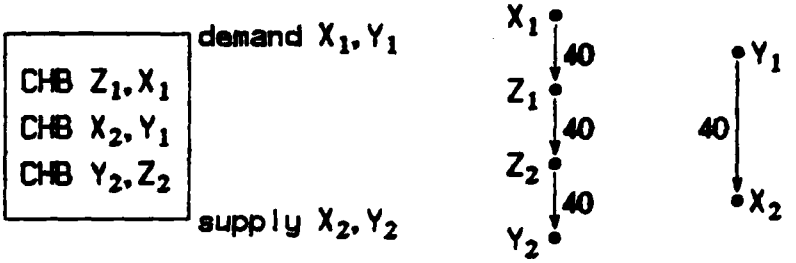


conflict relation from *R*

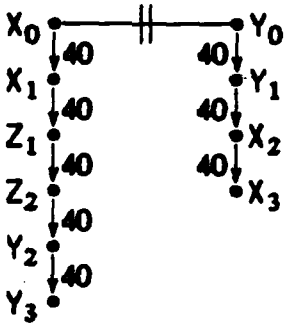
The absence of Z from the demand set means that Z is dead after the exchange

The most frequent arcs are compiled first, so say that the two arcs touching the

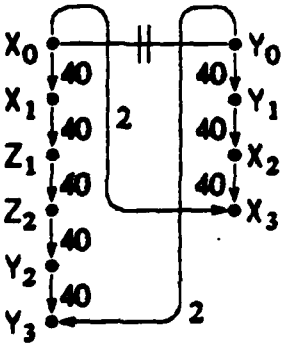
exchange box are compiled next. The first of these will cause the box itself to be compiled, resulting in cohabitation arcs and boundary sets below. CHB (cohabit) is a pseudo-op that provides a place for the occurrences. It requires no space in the eventual machine code, and no time to execute. The actual cohabitation information is in the graph on the right. (The number on a cohabitation arc denotes the cost of breaking it.)



Compiling the frequent arcs will result in nothing more than establishing cohabitation arcs among matching elements of supply and demand sets. This results in the following overall relations:

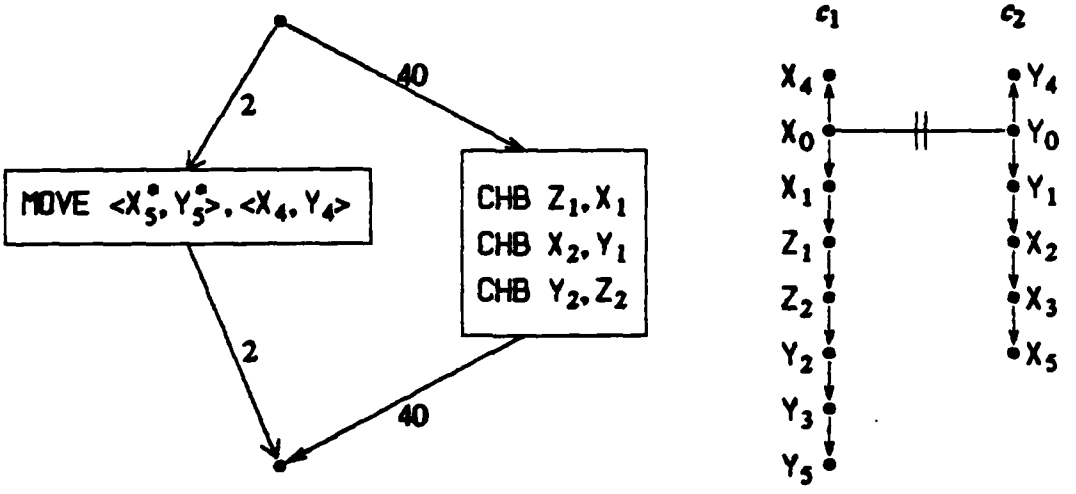


Note that no inconsistency has arisen yet. This happens when the remaining arc (with frequency 2) is compiled. The graph after compilation, but before resolution, is as follows:



It is clear that the cheapest way to resolve this is to break the X_0-X_3 and Y_0-Y_3 arcs.

This is done by move instructions placed on the infrequent arc. To avoid ordering problems, and to exploit machine instructions which move several quantities at once, it is convenient to postulate a "simultaneous move" instruction which we place on the arc. This results in the new flowgraph fragment and new relations (X_4, X_5^*, Y_4, and Y_5^* are *new occurrences for the new instruction*):



Remember that what actually gets "moved" is cohabitation classes, of which we presently have two, c_1 and c_2 as labeled above. Writing the simultaneous move in these terms, we have MOVE $<c_2, c_1>, <c_1, c_2>$. This begs to be compiled as an exchange instruction, if one is available. And note that the programmer wrote the exchange *on the other arc*!

4. Extra Occurrences

4.1 Motivation

There is a tradeoff regarding the construction of the cohabitation and conflict relation. On the one hand, it would be ideal if the mere selection of a set of arcs would indicate exactly where to place the moves to break the set. But there is so much choice in where to place moves that any data structure which would achieve this goal would be huge. On the other hand, if the relations don't give a reasonably good idea as to where the moves go, it means that as the relations are being constructed, one doesn't really have a very good idea of what the costs of breaking an arc are. This leads to either missed optimization opportunities (if over-estimation occurs), or extra expense in resolving inconsistences (if under-estimation occurs). Further, if the cohabitation and conflict information doesn't give a good idea about where to place moves, the calculation of precisely where to place them may be extremely expensive.

The compromise we use is an indirect one. Rather than associating with a cohabitation arc some kind of data indicating how to place the moves, we add a few extra occurrences of variables to the *flow*graph. These occurrences will appear in the cohabitation and conflict relations just like "legitimate" occurrences. The placement of these occurrences in the flowgraph is related to the likely places for inconsistency-resolving moves, so that they aid in determining where to place these moves. We noted earlier that cohabitation arcs are directed. The reason for this is to further aid in the determination of where to place moves.

4.2 Merge and Split Occurrences

Some extra occurrences of a variable may be added at points where, relative to the variable, flow splits or merges.

Definition. Given a variable V, a V-*merge node* is one which can be reached from different occurrences of V along paths whose intersection consists solely of the node *N*. A V-*split node N* is defined dually, i.e., is one from which distinct occurrences of V may be reached along paths which intersect only at *N*. □

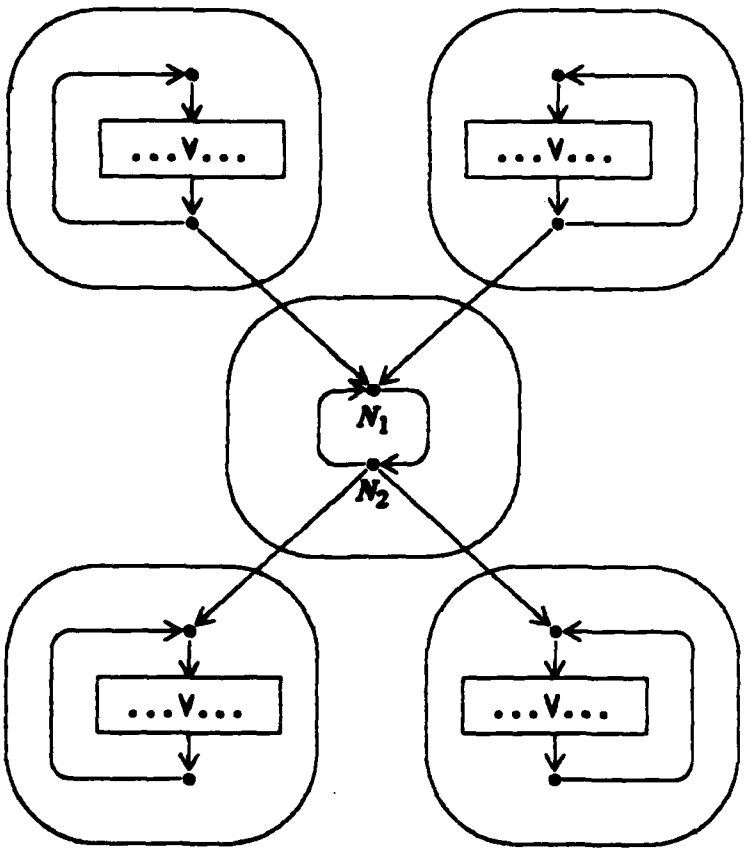
The calculation of V-merge and V-split nodes can be solved by using Tarjan's techniques for path-problems on directed graphs [6].

Not every V-merge and V-split node has an extra occurrence of V. The precise rules are:

M If *N* is a V-merge node and V is live at entry to *N*, there is an occurrence of V before the first change to a variable.

S If *N* is a V-split node and V is live at exit from *N*, there is an occurrence of V after the last change to a variable.

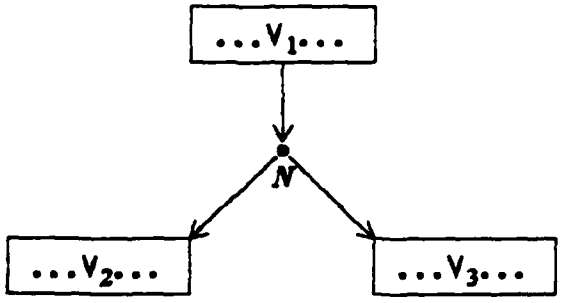
If necessary, extra occurrences are placed under the pseudo-op USE. The rationale for extra occurrences will become clearer as the techniques of cohabitation/conflict calculation, cost estimation, and move placement are discussed. We now give some hints as to why these seem to be the proper concepts. Consider the following program segment:



Suppose that the upper two uses of V are inconvenient to keep in the same place. The place to fix this is on one of the arcs coming into N_1 which forces V to be in

one place as the middle region is entered. The same remarks apply dually to the lower two uses of V and N_2.

As another example, consider the fragment:



Note that V is live at the bottom of N, so N is a V-split node, and an extra occurrence of V is placed at the bottom of N; call this occurrence V_4. In the compilation of the arc entering N, there would be a cohabitation arc from V_1 to V_4, which we denote $V_1 \rightarrow V_4$. Similarly, compilation of the arc exiting N on the right would produce the cohabitation $V_4 \rightarrow V_3$. Suppose it became necessary to break the cohabitation chain $V_1 \rightarrow V_4 \rightarrow V_3$. If each of the arcs exiting N has a non-zero frequency and flow is conserved, the arc leaving N has a lower frequency than the arc entering N. Thus the cost of $V_4 \rightarrow V_3$ is less than the cost of $V_1 \rightarrow V_4$, and the place to break the chain is $V_4 \rightarrow V_3$. This gives a good idea where to place the move—along the arc from N to the occurrence V_3. Without the extra occurrence one would have only the cohabitation $V_1 \rightarrow V_3$. If it must be broken, it is harder to figure out where to put the moves. It is also more difficult to come up with a general way of estimating the cost of breaking a cohabitation.

4.3 Intermediate Subgraphs

This section describes in more detail how the insertion of extra merge and split occurrences limits the part of the program involved when breaking a cohabitation. In order to discuss this, we assume that the program graph has a single entry and single exit node. We then use the standard graph terminology.

Definition. A node (or arc) N_1 *dominates* a node (or arc) N_2 when N_1 is on every path from the entry to N_2. Dually, N_1 *back-dominates* N_2 when N_1 is on every path from the N_2 to the exit. □

We wish to extend dominator terminology to occurrences. We do so by defining a *relation on paths and occurrences*, so that any graph-theoretic notion defined in terms of paths will extend to occurrences. We first need the notion of one occurrence lying *above* or *below* another occurrence in the same node (they may also be unordered). This is usually clear in any given context, and is not formalized further here. But using it, we have:

Definition. When we say that an occurrence o *is on a path from an occurrence* p to an occurrence q we mean both the following:

> If o is in the same node as p, it is below p; otherwise o is in some node on the path from the node of p to the node of q.
>
> If o is in the same node as q, it is above q; otherwise o is in some node on the path from the node of p to the node of q.

□

Note that if p and q are in the same node, this says that o is below p and above q. With this, we introduce an idea that is used throughout this paper:

Definition. Let p, q be any two occurrences (or nodes or arcs) of a program. The *intermediate subgraph* of p and q, written $G(p, q)$, consists of occurrences and arcs dominated by p and back-dominated by q, and of all nodes touched by the arcs.

□

We usually deal with intermediate subgraphs of p and q where p and q are different, but nearby, occurrences of the same variable. The live region of a variable partitions naturally into certain of these subgraphs, because of the insertion of extra occurrences. Before stating the main result, we have some preliminaries.

Definition. Let V be a variable of the program. A V-*free path* is one which has no occurrences of V in any of its nodes, except perhaps for the beginning and/or end occurrence, if the path begins and/or ends on an occurrence.

□

Definition. An occurrence V_1 V-*dominates* an occurrence (or node or arc) q if V_1 dominates q and if every V-free path from an occurrence of V to q begins at V_1.

□

In the next result and throughout this paper, we assume that any variables which are live at the entry to the flowgraph have an occurrence there (at least conceptually),

and that V-merge nodes are calculated accordingly.

Lemma 4.1 Suppose there is a V-free path from an occurrence V_1 to an arc A (resp. a first use o), and that V is live on A (resp. at o). Then V_1 V-dominates A (resp. o).
Proof. We consider the arc case, first showing that V_1 dominates A. Suppose there is some path from the entry to A which avoids V_1. This path must intersect the V-free path from V_1 to A at some node above A. Pick the nearest such node to V_1. This will be a V-merge node, because there are disjoint paths from distinct occurrences of V to this node. But a V-merge node must contain an occurrence of V, and by the V-freeness of the path, the node must be that containing V_1. But this contradicts the choice of the path to avoid V_1. The only possibility is that V_1 dominates A.

To show that V_1 V-dominates A, we must show that every V-free path from an occurrence of V to A starts at V_1. Suppose instead there is a V-free path to A from V_2. This path intersects with the path from V_1 to A in the hypothesis of this Lemma at a node above A, causing a contradiction as in the above paragraph. This proves the lemma in the arc case.

We next consider the lemma for a first use o. The proof here is essentially the same as for an arc, the difference being that when the paths intersect, it will be at a V-merge node, so an occurrence of V will appear above any first use in the node, in particular, above o. Thus, we do not have a V-free path from V_1 all the way to o. □

Definition. An occurrence V_1 V-*back-dominates* an occurrence (or node or arc) q if V_1 back-dominates q and if every V-free path from q to an occurrence of V ends at V_1. □

Lemma 4.2 Suppose there is a V-free path from an arc A (resp. a first use o) to a non-first occurrence V_1. Then V_1 V-back dominates A (resp. o).
Proof. Exactly dual to the proof of Lemma 1.
□
The main result of this section is this:

Theorem 4.1 Suppose the variable V is live on the arc A (resp. at first use o). Then A (resp. o) lies in a unique minimal intermediate subgraph of the form $G(V_1,V_2)$.
Proof. Look at any path from the entry to the exit through A or o. Let V_1 be

the last occurrence of V on this path before A or o, and let V_2 be the first occurrence of V on this path after A or o (there must be one, else V is not live on A or at o). By Lemma 1, V_1 dominates A or o, and by Lemma 2, V_2 back-dominates A or o; thus A or o is in $G(V_1,V_2)$. This proves the existence of V_1,V_2.

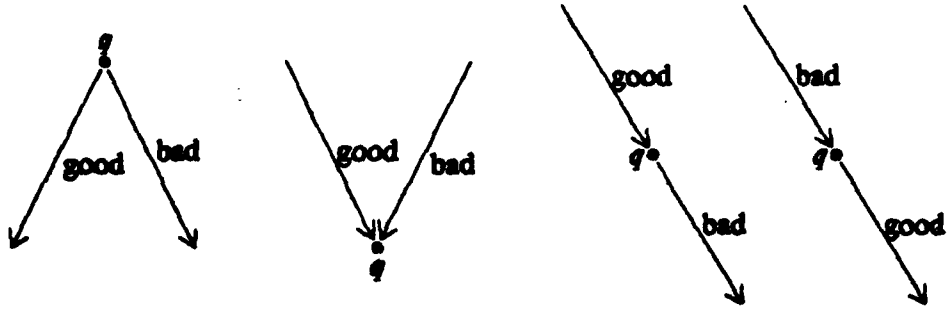
To prove uniqueness, assume A or o also lies in $G(V_i,V_j)$. By Lemma 1, any path from V_i to A or o must go through V_1. Thus $G(V_i,V_j)$ contains $G(V_1,V_j)$, and is strictly bigger if $i \neq 1$. By minimality, then, $i = 1$. Dually, we conclude $j = 2$. □

We note that this Theorem does not hold if the phrase "arc A" is replaced by "node N" or if we allow arbitrary occurrences o, rather than just first uses. For example, a V-merge node containing V_1 as the first use of V will be in $G(V_i,V_1)$ for several different i. On the other hand, V_1 itself does not belong to $G(V_i,V_1)$ for any i since it is not dominated by V_i; the same may be said for use occurrences above V_1 (but in V_1's node). While the partitioning of arcs and first uses is important in what follows, the lack of this for nodes and for general occurrences is not a problem.

A further consequence of the way that intermediate subgraphs divide up the flowgraph is the following.

Corollary 4.1 Let S be any connected subgraph of the flowgraph which has no occurrence of V, but in which V is live at some place. Then V is live in all of S and all of the arcs of S belong to the same minimal intermediate graph $G(V_1,V_2)$.

Proof. We use induction on the number of arcs. If this number is zero, the result is vacuous. If there is at least one arc, pick one, and by Theorem 4.1, choose V_1 and V_2. Suppose there is some arc for which the result does not hold. Either V is dead along the arc or the minimal intermediate subgraph is different, i.e., is of the form $G(V_i,V_j)$, where $i \neq 1$ or $j \neq 2$. By the connectedness of S, we may choose this "bad" arc so that it shares a node with some "good" arc whose subgraph is $G(V_1,V_2)$. There are four possibilities.

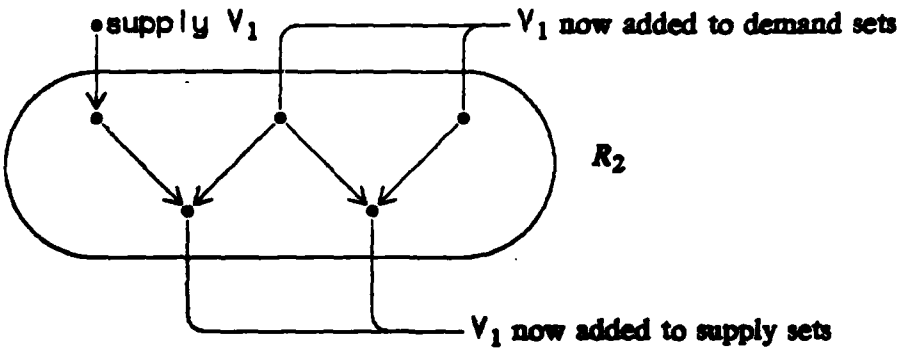


In the first case above, V is live on the bad arc because it has no occurrence at q. Since V_1 V-dominates the good arc, it must also V-dominate q; but if it V-dominates q it must V-dominate all the output arcs of q, in particular, the bad arc. Thus, $i = 1$ in this case. Now, suppose that the bad arc is not V-back-dominated by V_2. Since the good arc is, q is a V-split node, and must have an occurrence of V. But q is a node of S, and cannot have an occurrence of V. The only possibility is that the first case cannot arise. Dual considerations eliminate the second case as well.

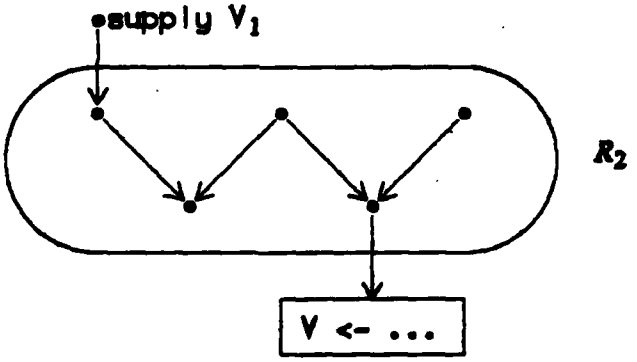
In the third case, since q has no occurrence of V, V must be live on the bad arc. Assume that $i \neq 1$. Then V-free paths from distinct occurrences converge on q, making q a V-merge node, the same contradiction as above. Dually, if $j \neq 2$, q is a V-split node, again a contradiction. This eliminates case three, and the dual argument eliminates case four. Thus there can be no bad arcs.

□

The importance of this corollary is that it controls the maintenance of supply and demand sets. We said earlier that an occurrence of a variable was in one of these sets if the variable was live at the appropriate point, and if it appears in the region. Suppose we are compiling an arc between regions R_1 and R_2, where R_1 mentions V but R_2 does not; assume that V is live along the arc being compiled. Then we expect to find V in the boundary set of R_1 but not in the boundary set of R_2. The importance of the Corollary is that merely be seeing an occurrence of V in the boundary set of R_1 and *not* seeing an occurrence of V in the boundary set of R_2, we know that V is live throughout R_2. We may thus put the occurrence on all of the entry and exit nodes of R_2, since these nodes will be entry and exit nodes of the newly combined region, except perhaps for the node touched by the arc being compiled. We give a picture of this "sideways" propagation:



The crucial role of extra occurrences in allowing this type of propagation is demonstrated by the following counter-example. The point is that without extra occurrences, the situation would look exactly like the above one, if attention is restricted to what is already in a region. However, in the example below, V is *not* live throughout the region, in particular, it is not live at the bottom of the lower right node. Thus we could not correctly add V_1 to the supply set of that node.

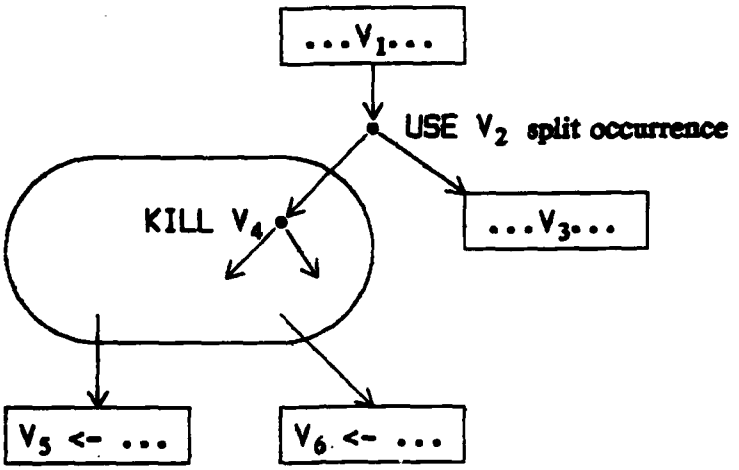


In the scheme we have proposed, R_2 would have split and merge occurrences of V added, so that there would be some demand occurrence of V at the node pointed to by the arc being compiled, before compilation of the arc begins.

4.4 Deadly Occurrences

It may happen that a variable in a program is live at the bottom of a split node, but dies out of one or more of the arcs of the node. In this situation, it is necessary to know in which direction the variable lives and which it dies. One possible way to solve this problem is to provide some data structure on arcs which yields this kind of information. However, rather than complicating things with a new type of data structure, we use extra occurrences. The idea is to let the variable live on all the arcs

out of the split node, and to kill the variable where necessary by placing an extra occurrence under the pseudo-op KILL. The picture is:



The KILL V_4 indicates to a forward scan that V is not needed—something that would not otherwise be known until the cut set of assignments is detected. This is one more way in which extra occurrences are used to aid later scanning. As we will see, no harm comes in assuming that V is live on the arc where it is really dead.

5. Compiling a Node

We consider the compilation of the smallest regions, i.e., nodes. Such a compilation produces a *kernel region*. The kernel regions are the repository of all machine code for the program. First we consider the input to this part of the code-generator. We have already assumed that the code-generator works from a graph representation of the program; in this chapter we will need to make some additional assumptions about the contents of the nodes, which together with the graph itself, constitute the *intermediate form* for the program. Each node will have a machine-independent, but nevertheless "primitive" operation. The earlier phases of the compiler may introduce temporary names, so that a statement from the source like W <- X*Y + Z will appear in the intermediate form as:

```
T <- X * Y          or          T1 <- X * Y
W <- T + Z                      T2 <- T1 + Z
                                W <- T2
```

The intermediate form will contain occurrences, so that if we had been following our usual notation, all the variables above would be flagged with subscripts. The code-generator will commonly appropriate the occurrences of the intermediate form for use in the machine code that it produces.

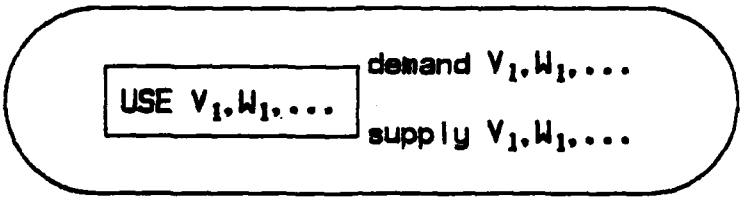
This chapter may be viewed as a further specification of what it means to "compile the node", as stated in the compilation algorithm (page 7). Although computation cost may ultimately force a special means for turning maximal flowblocks into regions, we consider here regions arising from single statements of the intermediate form. For each kernel region, we want:

> Boundary sets (remember that the node is both an entry and an exit node for the region).
>
> The cohabitation and conflict relations among all occurrences of the region.
>
> The "machine code" for the region (quoted, because it may not be exactly machine code and because it may contain pseudo ops).

Each of the sections of this chapter will consider certain types of kernel regions, and will provide invariant assertions about regions. The basis for the inductive proof of these assertions is that they hold for kernel regions.

5.1 Extra Occurrences

In this section we will treat the extra occurrences whose placement was described in the previous chapter. All of the extra use-occurrences at the top of a merge node (or bottom of a split node) are collected and placed under a single pseudo-op. In this case, the intermediate form simply becomes the "machine code" when the node is compiled. In the picture below, the rounded box represents the newly constructed region. It has one node, the rectangular box. Since this node must have entering and exiting arcs, necessarily not yet compiled, it will be an entry and exit node to the region, with the supply and demand sets as shown.



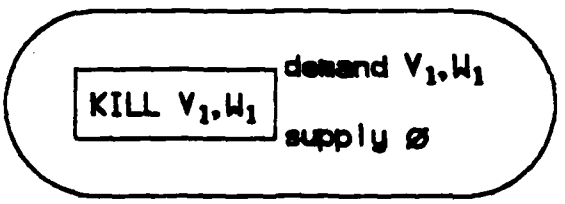
We now state explicitly the invariants describing precisely when an occurrence is in a boundary set.

BND If a variable is live at the bottom of an exit node, and if an occurrence of the variable appears anywhere in the region, then some occurrence of the variable will be in the supply set of the exit node; and conversely.

If a variable is live at the top of an entry node, and if an occurrence of the variable appears anywhere in the region, then some occurrence of the variable will be in the demand set of the entry node; and conversely.

The correctness of these invariants for the above kernel region follows because USE V_i appears only where V is live.

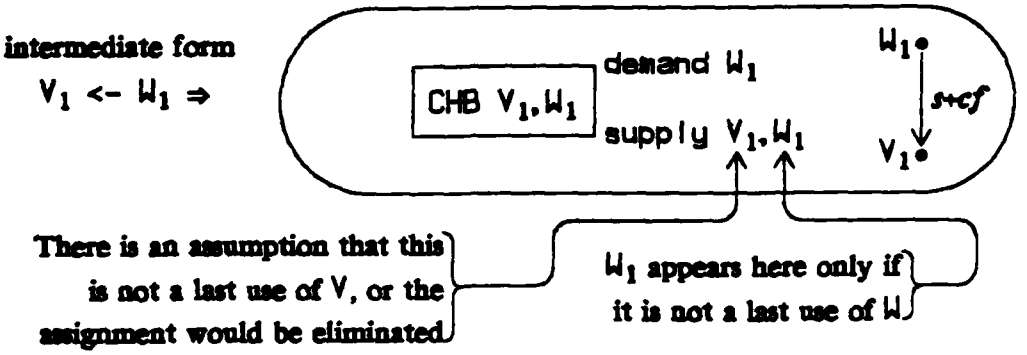
Killing occurrences are similar to use occurrences, but according to BND, they contribute only to demand sets:



This is the most trivial possible region.

5.2 Assignments

Recalling that we take an optimistic view of assignments (no code need be generated), we set up a pseudo-op to hold the occurrences and construct a cohabitation arc to indicate our assumption. Recalling the pseudo-op CHB, we have:



The cost to break this arc is indicated as $s + c \cdot f$, where s and c are the space-time factors for a move instruction, and f is the frequency of the assignment. The means of breaking this cohabitation arc is to change the CHB to some move opcode, thereby adding space to the code and time to the execution.

It should be noted that s and c are chosen optimistically. For example, on a register machine, they would be chosen on the basis of a register to register move instruction. If it becomes necessary to break the cohabitation, it might then be discovered (in the context of a by now larger region) that one or both of V_1 and W_1 are not registers. Thus the actual cost would turn out to be much larger than expected. As pointed out in section 3.3, this particular break in the cohabitation relation might not be used after all; if it is used, the cost of breaking it would be upped in light of the new knowledge.

The above kernel region has a single node, with associated supply and demand sets, just like the kernel regions of the previous section. It also has a cohabitation graph. Technically, so did the regions of the previous section, but those cohabitation graphs had no arcs, so we didn't discuss them. This non-trivial cohabitation graph provides an opportunity to introduce the invariant to be obeyed by the cohabitation graph of a region. As with the BND invariant, this invariant is formulated with respect to what can be seen in the region. We reason as follows. Suppose we begin an execution at some entry point to the region, and follow it to an exit node. The only

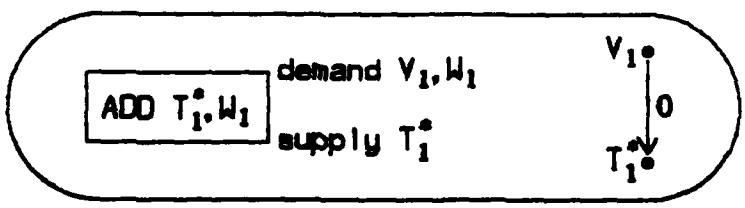
thing which can influence the execution is the values of the variables in the demand set of the entry node, plus any inputs obtained along the execution path. The only lasting effect of this execution path is outputs, and the values of the variables in the supply set of the exit node. The semantics of the source language will specify what execution path will be taken, given the values for variables in the demand set and inputs along the path, and will specify the outputs and the values of the variables at any point. In particular, at the exit node, the semantics will specify the values of the variables in the supply set. With this region-restricted view of correctness, we arrive at the following invariant for cohabitation.

> CHB The machine code for a region is correct for execution within the region, provided that all the occurrences in one cohabitation class are assigned to the same location.

We examine this invariant for the above region. The only demanded variable is W, and the semantics of the language prescribe that V and W have the same value after V <- W is executed. If V and W occupy the same location, they must have the same value upon exit. The invariant holds.

5.3 Computations

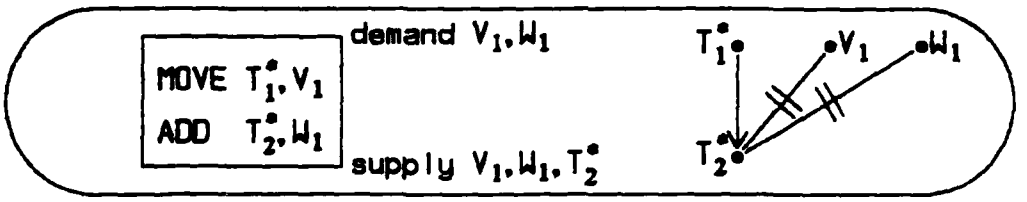
It is here that the most interesting cases arise. We shall consider as our fundamental example the source construct V + W. To make things interesting, assume that a machine add instruction has only two operands and always destroys one of them. In the case where V and W are last uses, we have the following compilation of T <- V + W (recall that * denotes a generation, so the instruction below adds the second operand to the first):



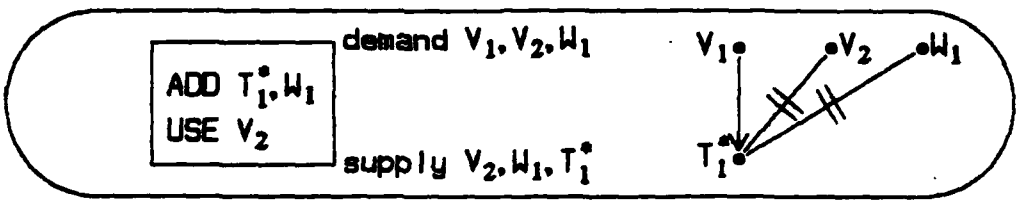
Since V_1 is a last use, we can make the optimistic assumption that V_1 and T_1^* can occupy the same location—thus leading to the cohabitation $V_1 \rightarrow T_1^*$. It is evident that the CHB invariant holds. Since V_1 and W_1 are last uses, and T_1^* a first use, we also see that the BND invariant holds.

The rationale for the 0 cost of the cohabitation is that conceivably, an inconsistency could be resolved by interchanging the two operands of the +, so that W_1 cohabits with T_1^*.

Next, let us suppose that neither V_1 nor W_1 is a last use. Then any simple ADD will destroy a needed variable. It might seem that a sequence such as the following is necessary:

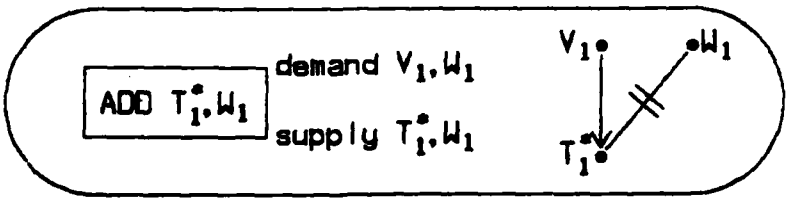


But, on some machines, such as the PDP-10, there are instructions which leave a result in two places or which can do a move while doing some other operation. Thus it might be possible to have two copies of a variable at this point, and to clobber only one. Since the strategy here is always to be optimistic, in this case we could generate as code only a simple ADD, demand two copies of a variable, and set up a conflict relation to indicate what the problem is.



The pseudo-op USE is employed to provide a place for a new occurrence of V. The costs on the cohabitation arc should be 0, because of the possibility that even if two copies of V cannot easily be made available, two copies of W can be. On a machine which does not freely generate copies, such a technique is of course not worthwhile. On a three operand machine such as the VAX, we could use all three operands: ADD_3 V_1, W_1, T_1^*. On many machines, the MOVE-ADD sequence will be the best possible code.

Finally, in the case that exactly one of the variables is a last use, we generate the following region (without loss of generality, assume V_1 is a last use, W_1 is not):



The cost of breaking this cohabitation arc depends upon what might be possible if this optimism doesn't work out—availability of multiple copies, three-operand instructions, or only a MOVE-ADD sequence.

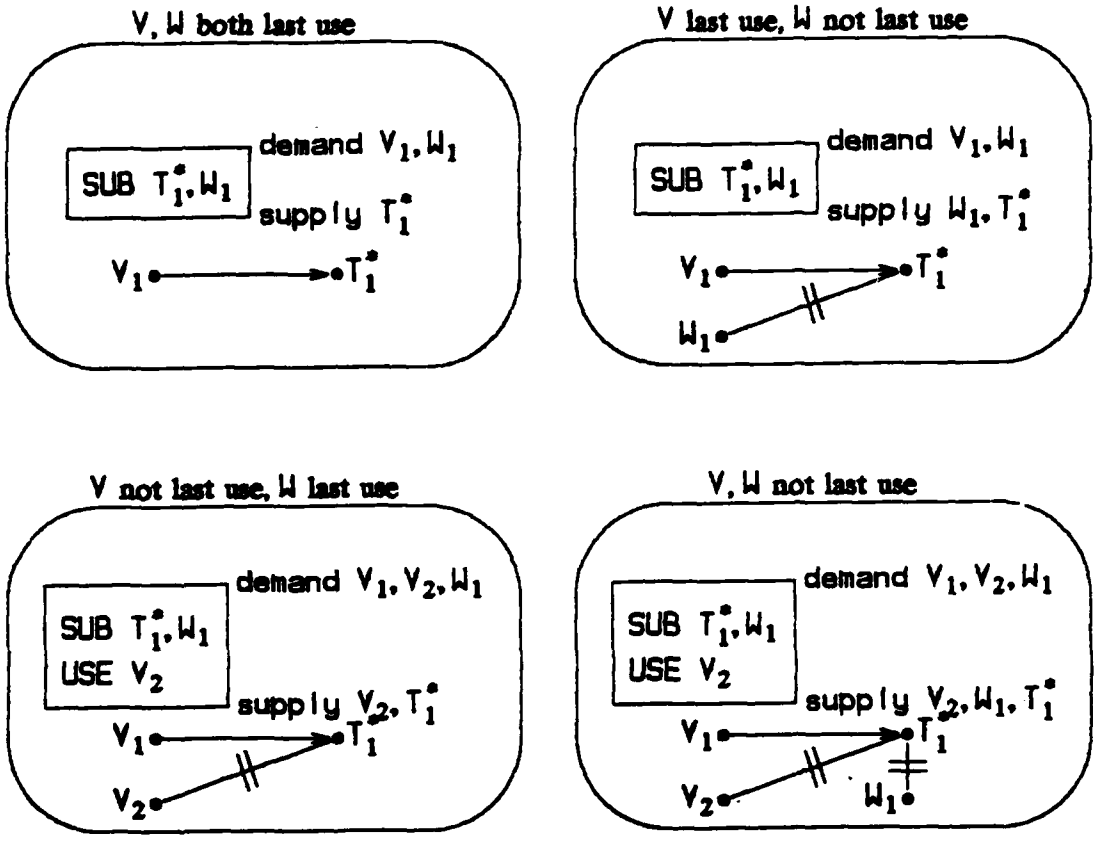
In the last three kernel regions pictured above, we have inserted a non-trivial *conflict* relation into the region, without saying precisely what correctness is. The invariant here is quite similar to CHB; it simply formalizes the notion that conflict prohibits excessive cohabitation.

> CON The machine code for a region is correct for execution within the region, provided that conflicting occurrences are not assigned to the same location.

Implicit in the invariant is that different cohabitation classes *can* be assigned to the same memory location (especially the same register), so long as they do not conflict. The correctness of CON for all the kernel regions of this section follows from a simple rule that has been observed in establishing kernel conflict:

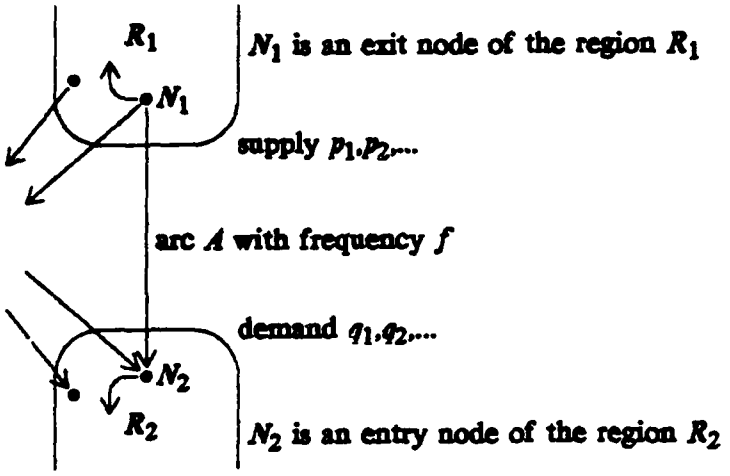
> Whenever an occurrence is both in the supply set and the demand set of a kernel region, it is in conflict with all the generations of that region.

We have chosen a commutative operation such as ADD to show how the basic scheme allows examination of the various ways of compiling the operation. We consider briefly the construct T <- V - W, where the machine has no reverse subtract instruction. The first attempt at compiling this is SUB V_1^*, W_1, regardless of whether V_1 is a last use. The difference between this and the ADD case is in the accompanying data structure, particularly the cost on the arcs. It always hurts to break $V_1 \rightarrow T_1^*$, because it cannot be done with a simple change to an instruction. (Ignore the possibility that it might be reasonable to calculate the negative of the desired quantity, and correct things later, as in U - V GT 0.) Here are the four cases, under the assumption that the machine can freely generate copies:


V, W both last use
SUB T_1^*, W_1
demand V_1, W_1
supply T_1^*
V_1
T_1^*
V last use, W not last use
SUB T_1^*, W_1
demand V_1, W_1
supply W_1, T_1^*
V_1
T_1^*
W_1
V not last use, W last use
SUB T_1^*, W_1
USE V_2
demand V_1, V_2, W_1
supply V_2, T_1^*
V_1
T_1
V_2
V, W not last use
SUB T_1^*, W_1
USE V_2
demand V_1, V_2, W_1
supply V_2, W_1, T_1^*
V_1
T_1
V_2
W_1

6. Compiling an arc

We finally come to the interesting part of compilation—compiling an arc. The following picture represents the generic situation.



In general R_1 may have other exit nodes, and N_1 may have other arcs leaving it, some compiled, some not; dually for R_2 and N_2. The occurrences supplied by N_1 (resp. demanded by N_2) are denoted p_1, p_2 ... (resp. q_1, q_2, ...). This does not mean that p is the variable of p_i. Rather, the notation $v(p)$ is used. Note that R_1 may be equal to R_2.

Our goal in this chapter is to devise the proper adjustments to the data structures (boundary sets, cohabitation and conflict relations, and machine code) so that after inclusion of the arc A all of the invariants of the previous chapter remain true. Naturally, we inductively assume the invariants for R_1 and R_2.

This chapter is an elaboration of what it means to "compile the arc" (compilation algorithm, page 7). The process of compiling an arc is driven by the members of the boundary sets, denoted above by p_1, p_2, ..., and q_1, q_2, ... The members of each of the two boundary sets divide into those that share a variable with an occurrence of the other boundary sets—these are called *matched* occurrences—and those that do not. In broad terms we have:

Algorithm 6.1 Arc-compilation

for $p,q \leftarrow$ matched occurrences along arc A
  Match p,q
for $p \leftarrow$ unmatched occurrences of N_1
  Propagate p throughout R_2
for $q \leftarrow$ unmatched occurrences of N_2
  Propagate q throughout R_1

The next two sections consider what it means to "propagate" an occurrence throughout a region, and what it means to match occurrences. Subsequent sections consider some derivative problems.

6.1 Propagation of an Occurrence

We consider the case in which the variable of some occurrence in the supply set is not the variable of any occurrence in the demand set, or dually, i.e., interchange the words "supply" and "demand". For simplicity, we talk about only the missing demand variable case, and do not continually make dual remarks for the missing supply variable case.

Using BND inductively, we conclude that R_2 has no occurrence of V, and consequently, that $R_1 \neq R_2$. Thus the new region will be $R \triangleq R_1 \cup \{A\} \cup R_2$. Then we invoke Corollary 4.1 and deduce that V is live throughout R_2. The combined region R is now in danger of violating BND, since it now has an occurrence of V (namely V_1), but no occurrences of V in the boundary sets of what used to be R_2. These observations prove that the following step maintains the correctness of BND for the variable V.

Algorithm 6.2 Propagation of an occurrence, step 1

Add V_1 to each boundary set of R_2.

This rule is the basis for the term "propagation." Aside from initial elements of boundary sets from the construction of kernel regions, this is the only way that these sets grow.

We next consider what we must do to maintain the CHB invariant, relative to the variable V. The answer is, nothing at all. To see why, consider an execution path in R. Since $R_1 \neq R_2$, such a path will lie entirely within R_1 or R_2 individually, or will cross the arc A exactly once. Since one of the original regions has no occurrence of V, there are no new requirements relating cohabitation of occurrences of V. Thus, no

change to the cohabitation graph, at least on V's behalf, is necessary.

The situation is quite different when we consider the CON invariant. To see why, let us suppose that V appears in R_1 but not in R_2. Let us assume that there is another variable W, which might or might not appear in R_1, but in any case does not have a generation there. An execution starting in R_1, entering R_2 (via A), and passing through the generation of W can change the value of V. But since V is live at exit nodes of R_2, the execution path might not have the correct effect on the value of V. In order to maintain CON, we must add some conflict relations. Let V_1 be the occurrence of V.

Algorithm 6.3 Propagation of an occurrence, step 2

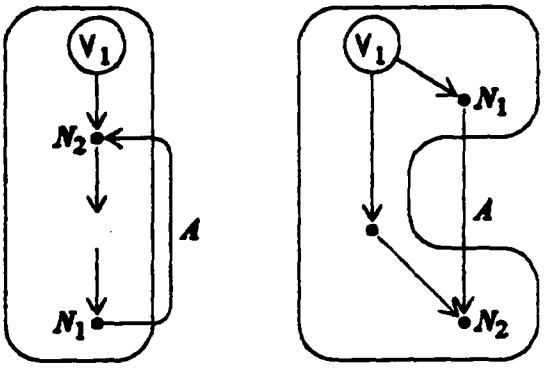
Establish conflict between V_1 and every generation of region R_2.

To show that CON holds for the region R in light of the above action, we follow the above reasoning which we used to motivate the action, except we can now observe that no change to a location in R_2 can affect V, so that the value of V as an execution path crosses arc A (inductively correct from R_1) is the same value that it has at an exit node of R_2. The correctness of CON is the last step in showing the correctness of all the invariants, relative to V, when V_1 is an unmatched occurrence. Note the similarity of this algorithm and the rule for kernel conflict, page 33.

Because of the way that conflict is created for unmatched occurrences, the set of such occurrences is called a *brush set*, the image being that these occurrences "brush" over the region, creating conflict. It is a simple matter to go through R_2 establishing conflict between its generations and V_1. It is also simple to see that doing this naively is going to be very expensive as regions get large. It would have to be done for every unmatched occurrence of the supply set, so the number of steps would be proportional to the product of the size of the brush set and the number of generations of the region. Fortunately, something much more efficient than a literal interpretation can be achieved; the algorithms and data structures are discussed in sections 6.6 and 6.7.

6.2 Matching Occurrences

In this section, we consider a pair of matched occurrences, i.e., p and q with $\nu(p) = \nu(q)$. It is possible that $p = q$, because of a loop or because of sideways propagation, as pictured below ($p = V_1$, A is the arc being compiled).



There is no action to be performed in this case, but we must of course prove that the invariants hold after inclusion of A in the region. Observe that we must have $R_1 = R_2$, since the same occurrence cannot be in separate regions. The BND invariant follows directly from that. Consider an execution path in $R \triangleq R_1 \cup \{A\}$. We use induction to prove correctness up to the *first appearance of* A, then observe that the value of V is delivered at N_1 in the location of the cohabitation class of p, and is required at the same place at entry to N_2. Thus, relative to V, correctness extends across A, and we may repeat this argument till we get to the end of the execution path.

If p and q are distinct, we write $p = V_1$ and $q = V_2$. In this case, R_1 and R_2 may or may not be the same, but in either case, there is no need to change the boundary sets inherited from R_1 and R_2. The BND invariant holds by direct induction. However, CHB does not necessarily hold, because we must make sure that V_1 and V_2 are assigned the same memory location, or we cannot make a correctness argument for an execution path crossing arc A. The purpose of the following algorithm is to formally state what must be done to the cohabitation graph to maintain CHB.

Algorithm 6.4 Match occurrences (for one variable)

Establish a cohabitation arc between V_1 and V_2.
If there is an inconsistency then call the inconsistency resolver.

If we can perform this part of the algorithm without getting an inconsistency, then CHB follows for the usual reason—we can prove correctness across the arc A. To

prove CON, note that occurrences which conflict before the union will necessarily conflict after the union. Thus the statement of CON for the new region is logically weaker than the conjunction of CON for the original regions, and it holds.

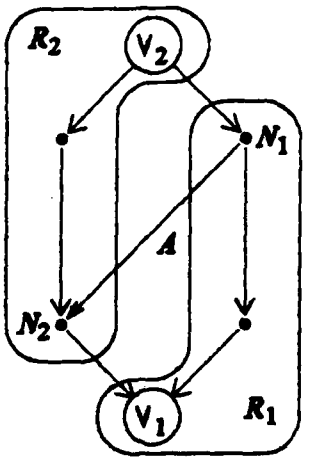
If the above step results in inconsistency, then to maintain CHB and CON we must modify code, as well as the cohabitation and conflict relations. This is the subject of chapter 7.

It may have gone unnoticed that the remarks of this section apply to the case in which the variable V_2 is an extra occurrence under a KILL pseudo-op. It may well be the case that V_2 is the only occurrence of V in R_2. The fact that V_1 matched V_2 means that V_1 did not brush over R_2, so no conflict was established between V_1 and any generation of R_2. Further, the fact that V_2 is not a generation means that no conflict was established between it and any brush occurrence from the supply set of N_1. Thus, the occurrence V_2 is an innocuous bookkeeping device, as we claimed earlier.

6.3 Establishing a Cohabitation Arc

The algorithm for matching occurrences in the previous section begs the question of how to establish the cohabitation arc between V_1 and V_2. There are two problems: which direction does the arc go, and what is the cost of breaking it?

We first consider the question of direction. From the way regions are constructed, it is clear that there is a V-free undirected path from A to V_1 and from A to V_2. If boundary sets contain merely occurrences, there is no way to know whether A is in $G(V_1,V_2)$ or $G(V_2,V_1)$. However, boundary sets are *disjoint* unions of subsets of boundary sets of kernel regions, and it makes sense to say that an occurrence *in a particular boundary set* was originally in a supply set or originally in a demand set. (Because an occurrence can appear in both boundary sets of a kernel region, it may have different origins with respect to different boundary sets.) This information is simple to keep track of, and tells which direction the cohabitation arc must go. The following picture makes this clear, and also shows that cohabitation arcs can point in the direction opposite to arc A.



During the compilation of R_1, the occurrence V_1 starts in the demand set of the bottom node, and eventually appears in the supply set of N_1; dually, V_2 will appear in the demand set of N_2. If we make the rule that the cohabitation arc goes from "kernel supply set" to "kernel demand set", the arc direction will be from V_2 to V_1, consistent with the direction of flow. Put differently, we observe that A is in $G(V_2,V_1)$, not $G(V_1,V_2)$. Formalizing this, we have:

Algorithm 6.5 Establish a cohabitation arc

```
if V1 was originally in a supply set
   if V1→V2 does not already exist
      make an arc V1→V2
else (V1 was originally in a demand set)
   if V2→V1 does not already exist
      make an arc V2→V1
```

This guarantees the following characterization of established cohabitation arcs:

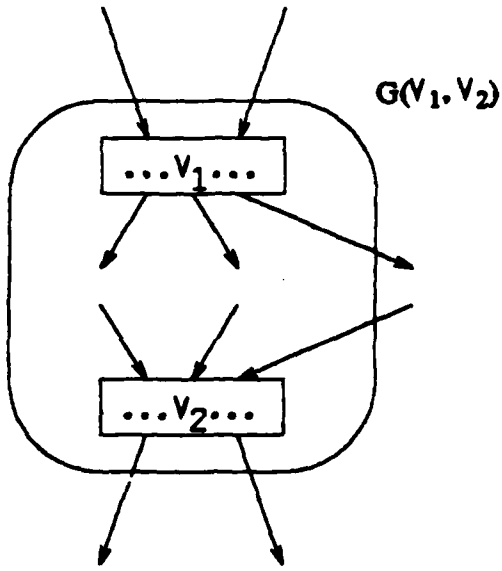
Theorem 6.1 Suppose there is a directed V-free path from V_1 to V_2. There is an undirected V-free path from V_1 to V_2 lying in some region and in $G(V_1,V_2)$ if and only if there is a cohabitation arc from V_1 to V_2.

Proof (Referring to the above picture may help, except that the labels V_1 and V_2 are reversed.) The backward implication is vacuous before compilation begins, because there are no regions, and no cohabitation arcs. Since creation of a kernel region does not add any new undirected V-free paths, we must merely insure that it does not create any cohabitations between different occurrences of the same variable. This would happen only for an assignment of the form V←V; we assume these have been eliminated. We next consider the compilation of an arc A from N_1 to N_2 which

results in the establishment of a cohabitation arc from V_1 to V_2. In this case the algorithm for establishing cohabitation arcs says V_1 was originally in a supply set, and V_2 was originally in a demand set. Now, step 1 of the algorithm for the propagation of an occurrence is the only way that occurrences are propagated, so the only way that V_1 could be in the supply set of N_1 or the demand set of N_2 is if there is an undirected V-free path in the respective region; dually for V_2 and the node at the other end of A. The paths together with the arc A constitute a connected subgraph with no occurrence of V, and V is clearly live along A. By Corollary 4.1, A and each path must lie in $G(V_1,V_2)$, providing the required undirected V-free path. It will lie in the region created by compiling arc A, and so meets all the requirements. Hence, if a cohabitation arc exists, the path exists.

Conversely, suppose we compile an arc which completes the first undirected V-free path of compiled arcs from V_1 to a distinct occurrence V_2, where the path lies in $G(V_1,V_2)$. Then there must be an occurrence of V_1 in one of the sets attached to the entry or exit node, and an occurrence of V_2 in the other; V_1 will necessarily be in its kernel region's supply set, and V_2 in its kernel region's demand set. Further, by inductive use of the theorem itself, there is no cohabitation arc from V_1 to V_2. But this is precisely the condition under which such an arc is established, by the above algorithm. Thus, if the path exists, the arc exists. □

The other problem that this section considers is obtaining a good lower bound on the cost of breaking the cohabitation. Breaking is done by some kind of move instruction. The first thing to determine is how often a move must be done, which is related to the part of the program in which the move can be placed, namely, the intermediate graph $G(V_1,V_2)$. Consider the nodes of $G(V_1,V_2)$ which do *not* have an occurrence of V. All arcs incident upon these nodes are in $G(V_1,V_2)$, as is clear from Corollary 4.1. The general picture of $G(V_1,V_2)$ is thus:



To determine how often a move must occur we sum the frequencies of all the arcs leaving the V_1 node which are in $G(V_1,V_2)$, or equivalently, the sum of the frequencies of arcs in $G(V_1,V_2)$ which point to the V_2 node. This gives the transmission frequency from V_1 to V_2, denoted $f(V_1,V_2)$.

If the node of V_1 has a single outgoing arc, or the node of V_2 has a single incoming arc, $f(V_1,V_2)$ is easy to calculate, namely, it is the frequency of that single arc. Even in the general case it is not exceedingly expensive to calculate $f(V_1,V_2)$; nevertheless, it may be the case that it is advantageous to use f, the frequency of the arc being compiled, to serve as a lower bound for $f(V_1,V_2)$. This is justified by the following result.

Lemma 6.1 When establishing a cohabitation arc between V_1 and V_2, the frequency f of the arc being compiled is less than or equal to $f(V_1,V_2)$.

Proof. By the compilation order of arcs, we know that f is a minimal value along some undirected V-free path between V_1 and V_2. Clearly, this minimal value cannot exceed $f(V_1,V_2)$, since some arc touching the V_1 (or V_2) node will be on the path, and any arc touching this node cannot have a frequency greater than $f(V_1,V_2)$. □

Using f to approximate $f(V_1,V_2)$ is in accord with our underestimation philosophy. Only experience will tell if this results in a decrease in compile time, but we assume it will.

Once we have a frequency, we try to estimate the cost. Suppose we break the cohabitation with a single move instruction. We would choose s and c for this instruction optimistically but realistically—if V_1 and V_2 are already known to be not in registers, for example; s and c reflect this knowledge. Given s and c, the cost is $s + c \cdot f$.

What this analysis does not take into account is that it may not be necessary to insert a brand new move instruction. Instead, it might be possible to achieve the move by some trick; for example, on the MC68000 there is a single instruction which will push any subset of registers onto the stack. In this case, the s parameter becomes 0 (because no extra code is required), and the c parameter becomes the incremental time cost of pushing one more word. Thus, the cost of breaking the arc becomes $c \cdot f$. If copies can be freely generated, it may be possible to break the cohabitation with s and c both 0.

This trickiness in moving quantities presents a dilemma. One has the feeling that very often, the tricky instruction which breaks the arc with the estimated (minimal) cost is not going to be an option. If the code-generator over-estimates the cost, it may miss a chance at an optimization. If it under-estimate the cost, it may spend a lot of time in the inconsistency resolver looking futilely for trickiness that does not exist. Once a code-generator is in operation, it will be possible to gain some experience in how to make this compromise. There is even the possibility of being able to dynamically adjust its "optimism" in assigning a cost. The maximum possible gain is the cost of a naive fix minus the cost of a subtle fix, e.g., $s + c \cdot f$ for a move instruction minus, say, 0 for freely generated copies, or a different $s + c \cdot f$ for some more expensive instruction variant. The cost of looking for a subtle fix will be roughly proportional to the size of $G(V_1,V_2)$. With experience, we can learn how often optimism pays off, and how to estimate the actual cost of searching. The code-generator can adjust its under-estimation so that the expected payoff exceeds the expected cost.

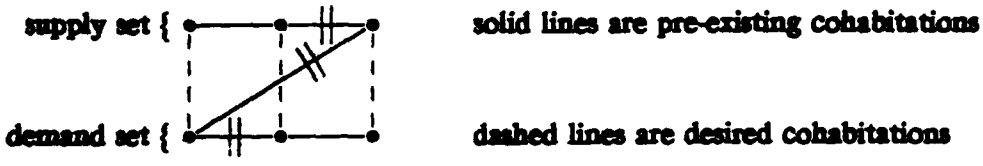
6.4 Inconsistency Detection

We maintain the consistency of cohabitation and conflict relations when compiling arcs. There are two aspects to this: first, detecting whether an inconsistency would arise if the arc were compiled without any modification of existing code, and second, if such an inconsistency would arise, modifying the code so that consistency is maintained. This section discusses the detection problem; the resolution problem is described in the next chapter.

Thus far, the only representation that has been discussed for the cohabitation relation is its (labeled and directed) graph. If this is the only representation of cohabitation, detection is very expensive, because to determine cohabitation, we must enumerate members of a cohabitation class (via the graph itself). We provide an oracle for rapidly determining whether two occurrences are in the same cohabitation class, given that cohabitation classes are continually combined by new cohabitation arcs. We use the standard FIND/UNION technique for this purpose (see [7]). Note that it is occasionally necessary to tear up a cohabitation class into two pieces, during inconsistency resolution. The updating of this structure when doing so is straightforward, but proportional to the size of the class. Since we assume that breaking up large cohabitation classes is rare, and since the cost of doing so is at least proportional to the size of the class, we assume the expense is worth it.

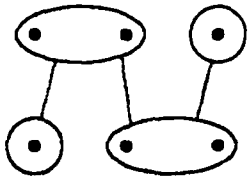
To understand the data structures used in inconsistency detection, we examine more closely the circumstances under which the relations change. The simplest case occurs in intra-region compilation. Since there are no brush occurrences in this case, no new conflict is generated. Assuming inductively that the region has no inconsistencies before the arc is compiled, the only way that one could arise is because of the addition of the new cohabitation relations required when matching occurrences. If Algorithm 6.2 says to establish an arc between occurrences which already cohabit (a question for which we now know how to compute an answer), there is no possible problem. But if the two occurrences do not cohabit, they may conflict and what we want to know is whether any two occurrences of the respective cohabitation classes of the ends of the arc are in conflict. This suggests the concept of the conflict of cohabitation classes or *class conflict*, where the rule is that cohabitation classes conflict if any of the occurrences in the respective classes are in conflict. We shall not immediately discuss the implementation of this relation, but,

assuming an oracle for it, we will look more closely at the interplay between the various cohabitation relations we want to establish. What we do not want to do is to establish a cohabitation which will have to be broken a few steps later. To see how this can happen, consider the following example:



Assume that the only conflict among the six points is that indicated, and that no other occurrences cohabit with these. If we try to establish the middle arc first, there is *no conflict* preventing it. If we next attempt either of the other cohabitations, an inconsistency will arise. If this second arc is more expensive to break than the middle one, then we would break the middle one—the one we just established. But if it is cheaper to break, or even the same, we would "break" this cohabitation before it really was ever established. The same would happen with the third arc. Now, assume all arcs are equally expensive. If there is no way for the resolution machinery to re-establish a cohabitation arc (and remove the code used to break it), then in the case where all cohabitations are equally expensive to break, we wind up with two broken arcs, and one established. But if we had waited and considered all three arcs at once, it would be clear that the best approach is to break (i.e., not establish) the middle arc, and to establish the others. Even if the resolution machinery is capable of reconsidering its placement of moves, it is more work than doing things right the first time.

In order to gather up all the cohabitations which are to be considered together, we look ot the cohabitation classes of the occurrences in the boundary sets. Just as we defined the idea of class conflict by raising the conflict relation to cohabitation classes, we define *class matching* by raising the matched relation to cohabitation classes: that is, two cohabitation classes are *matched* if each contains an occurrence such that the pair of occurrences is matched along the arc being compiled. Then we form the graph of this relation, calling it the *class graph*, since its nodes are cohabitation classes. The picture below is the class graph for the relations of the previous picture, with cohabitation classes encircling their individual occurrences, and the lines indicating class matching:



The connected components of the class graph, called *class components*, will be in distinct cohabitation classes after compilation of the arc. (A cohabitation class with no occurrence in a boundary set will be a class component unto itself.) The purpose of this construction is:

Definition. In a class component, a *match inconsistency* arises if two (cohabitation) classes are in class conflict in the original region.

□

Using this definition, we revise the algorithm for matching:

Algorithm 6.6 Match occurrences (intra-region case, all variables)

```
for each class component
    if there is match inconsistency in the component
      Call the inconsistency resolver on the component
    else
      Establish the cohabitation arcs for the component
```

Theorem 6.2 Cohabitation arcs added by the above step do not cause an inconsistency, and conversely, the inconsistency resolver is given a problem only when the addition of the cohabitation arcs would cause an inconsistency.

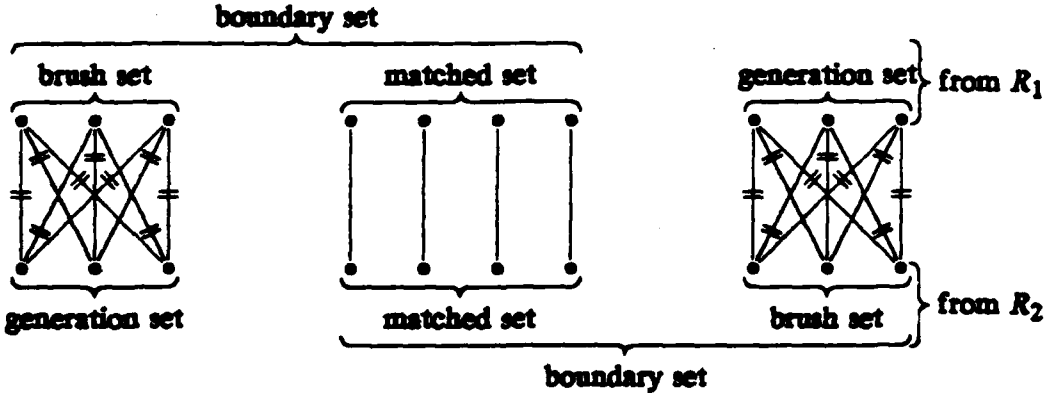
Proof. To start the induction, we use the fact that the relations of the region are consistent before compilation of this arc. Inductively assuming consistency, consider adjoining all of the arcs of a class component. Then all of the old cohabitation classes wind up in the same new cohabitation class, because of the connectedness of the class graph by arcs just added, and because of pre-existing cohabitation of the elements of old cohabitation classes. Suppose that two of the old cohabitation classes involved in this merge had pre-existing class conflict; then an inconsistency would arise. Thus the algorithm gives problems to this inconsistency resolver only when an inconsistency would actually arise. Conversely, if an inconsistency cycle arose, it has to involve conflict of old cohabitation classes in the class component, since these are the only ones that are merged, and since no conflict arcs are added.

□

Thus, we have reduced the problem of inconsistency detection to the problem of

knowing which classes are in conflict before compilation of the arc (which we have yet to discuss) and to an oracle for cohabitation, namely FIND. In the case that we establish an arc in the cohabitation graph between previously disjoint cohabitation classes, we also do a UNION of the cohabitation classes, so that FIND knows about the new cohabitation relation; we must ensure that the class conflict oracle knows about the cohabitation class merge—this is discussed in section 6.6.

We next consider inter-region compilation. As usual, we assume that the constituent regions have consistent relations; since the sets of occurrences of the two regions are disjoint, any inconsistency must involve at least two of the new relations created when compiling the arc in question, and at least one of them must be a cohabitation arc. This is because an inconsistency is a cycle of relations, of which precisely one is conflict, the rest being cohabitations. This cycle must involve at least one occurrence in each of the constituent regions, so must cross from one region to the other at least twice. Moreover, realize that when compiling an inter-region arc, the boundary sets in question are partitioned into brush sets and matched sets. The following picture is useful in visualizing the relations:



As in intra-region compilation, we do not try to add one relation at a time. Rather, we look at cohabitation classes of the occurrences in the boundary sets—remember, the regions are previously disjoint. As before, we form the class graph on the cohabitation classes, where the arcs are induced by the desired (but not yet established) cohabitations due to the arc being compiled; this time the graph is bi-partite. In each of the connected components (again called class components) of the class graph, we can determine whether any inconsistency will arise involving only new cohabitations by the same definition of match inconsistency. The question of

class conflict has to be asked only of cohabitation classes in the same constituent region, and therein lies the elegance of match inconsistency in inter-region compilation—usually a class component has at most one class in each region, so that there is no necessity to ask any questions of the class conflict oracle.

There is only one remaining way in which an inconsistency can arise: its cycle must involve new conflict.

Definition. In a class component, *brush-generation inconsistency* arises if the component has in one region a class that contains a brush occurrence, and in the other region a class that contains a generation.

□

This leads to the rest of the revised algorithm for matching.

Algorithm 6.7 Match occurrences (inter-region case, all variables)

```
for each class component
    if there is match or brush-generation inconsistency
        Call the inconsistency resolver on the component
    else
        Establish the cohabitation arcs for the component
```

Theorem 6.3 During inter-region compilation, the algorithm gives class components to the inconsistency resolver precisely when inconsistency would arise by establishing the desired relations.

Proof. We have already seen that if an inconsistency arises, it involves either two new cohabitations or a new cohabitation and a new conflict. In the first case, Theorem 6.2 states the desired result, with the proof differing only in how the induction is started. Here, the initial relations for the (combined) region are the unions of the relations on the constituent regions. Since the relations are consistent on their respective regions, and since the domains are disjoint, this initial relation is consistent, and the induction started.

In the second case, it is clear that an inconsistency cycle having new conflict from a brush occurrence o to some generation of the other region must go through at least one cohabitation added while compiling this arc, and must contain an occurrence in each of the constituent regions. The cohabitation classes to which o and the generation belong are thus matched, and so are in the same class component. This is precisely the condition that there be a brush-generation inconsistency. Conversely, if

there is brush-generation inconsistency, it is clear that an inconsistency cycle exists, so no non-inconsistencies are sent to the resolver.

□

Note that the detection of brush-generation inconsistency avoids any use of the class conflict oracle, using only questions about cohabitation and whether cohabitation classes have generations. Since cohabitation classes are always formed by the union of cohabitation classes, it is easy to keep track of whether they have generations.

In summary, this section has reduced the problem of inconsistency detection to that of two as yet undescribed oracles:

Given two cohabitation classes, are they in conflict?

Given a node, enumerate the elements of its supply or demand set.

The emphasis has been on minimizing use of the class conflict oracle, which we were able to do particularly well for inter-region compilation.

6.5 History Trees and Cohabitation Classes

This section describes data structures that are central to the implementation of both the class conflict oracle and the enumeration of boundary sets. The *history tree* is a record of region merges. The leaves of this tree are kernel regions. Each internal node corresponds to the compilation of an inter-region arc; such nodes have two descendants, corresponding to the constituent regions. We use the direction of the arc being compiled (A) to distinguish the left descendant (which A leaves) from the right descendant (which A enters). Each node has a pointer to its immediate ancestor; the top node of a history tree will of course have a nil pointer to its immediate ancestor. Because of the scattered nature of compilation, there will be a forest of history trees.

Suppose we have two kernel regions. They are in the same region precisely when they are in the same history tree. Moreover, if they are in the same history tree, their least common ancestor corresponds to the arc-compilation which first put them in the same region. As we shall see, the least common ancestor algorithm plays a role in the class conflict oracle. It becomes natural to think of a node of a history tree as corresponding to a time in the compilation process—nodes higher in the tree happen after nodes lower in the tree. It will turn out to be important for each

cohabitation class to have a pointer to a node in the history tree, corresponding to the time at which it was formed (section 6.6).

The representation that we use for cohabitation classes is similar to that of the history tree. Cohabitation classes either come directly from kernel regions or are the unions of other cohabitation classes. In the implementation of the class conflict oracle, it is necessary to know whether the union arose from an inter-region or intra-region compilation; a bit can record this information. It is necessary to know *when* the union occurred, that is, each cohabitation class has a pointer to a node in the history tree; this is called its *formation time*. It is also necessary to be able to retrieve the cohabitation classes whose union formed a cohabitation class; thus a cohabitation class, like a node in the history tree, has pointers to two descendants. In intra-region unions, the order of the descendants is irrelevant. But in the inter-region case, as in the history tree, we use the direction of the arc being compiled to determine a left and right son. Thus, for these unions, we know that the left sub-cohabitation class lies in the left sub-region of the corresponding formation time in the history tree. In inter-region compilation, if a class component has several cohabitation classes in the same constituent region, these are gathered up first by intra-region unions; the final step in forming the cohabitation is one inter-region union. The rationale for this becomes clear when the class conflict oracle is discussed.

The previous paragraph presents all the fields of the cohabitation class data structure that are necessary for the class conflict oracle. We also want to be able to ask whether two occurrences cohabit. For this, we use upward pointing arcs, and the standard FIND/UNION machinery.

As we saw in the previous sections, brush sets play an important role in compiling an arc. It is convenient to record the two brush sets involved in an inter-region arc compilation—one from the supply set, one from the demand set—in two fields in each node of the history tree. Although brush sets can be of arbitrary size, all that is recorded is a pointer (in section 6.7 we will discuss efficient representation of the brush sets themselves). Thus, the size of the history tree grows linearly with the size of the program.

In the next two sections we will propose other fields for history tree nodes and

cohabitation classes. The extra space for these fields buys time. The fields described in this section are a minimal set.

6.6 The Class Conflict Oracle

There are only two operations of interest on the class conflict relation; we want to know whether two (cohabitation) classes are in conflict, and we want to coalesce classes in it, where the classes are not in conflict. There are several representations which one might use to achieve this. We discuss several which have been tentatively discarded. The most obvious is a bit matrix representation, which has the advantage that one can determine quickly whether conflict exists. The disadvantage here is that space grows irrevocably as the square of the number of cohabitation classes and that coalescing two classes requires a number of operations proportional to the total number of classes in the class conflict relation. Another idea is to use a direct implementation of the graph of the relation. This has the disadvantage that determining conflict (adjacency of two nodes) has a cost proportional to the minimum of the degrees of the nodes, and that the space cost will be proportional to the number of arcs, which in practice will probably grow as the square of the number of cohabitation classes, and with a non-negligible constant of proportionality. This method does have the advantage that coalescing of nodes can be done in constant time, given the proper representation and the willingness to live with a few redundant arcs. Increasing the time slightly can save the space of the redundant arcs, if this is deemed necessary.

The solution we propose for the class conflict oracle is based on intimate knowledge of the way that conflict of occurrences, and thus of cohabitation classes, arises. As we said in the previous section, the history tree is the key data structure. What does the history tree have to do with conflict and cohabitation? Because of the compilation process, conflict arises either in kernel regions or in the merging of disjoint regions—never in the compilation of an intra-region arc. The way that the history tree is used to determine class conflict has the following outline:

> Find the least common ancestor (lca) in the history tree of the formation times of the cohabitation classes in question. If these classes conflict, it is either because at least one of them was formed at the lca, and the constituent parts were previously in conflict, or because a conflict arc was formed at the lca, and one of the classes had a brush occurrence in one of

the regions while the other class had a generation occurrence in the other region.

We now give some specifics on the implementation of the class conflict oracle. Let the classes be c_1,c_2 and their respective formation times be ft_1,ft_2, and their least common ancestor be t. After t is calculated, the first case breakdown is on whether either of ft_1 or ft_2 is equal to t. Consider first when neither is, where we argue as follows. No conflict could exist between c_1 and c_2 before t, since those classes then lie in disjoint regions. On the other hand, no conflict will be added between c_1 and c_2 after t, since conflict is not added once occurrences are in the same region, and since neither cohabitation class expands after t (use definition of formation point and history tree). Thus, c_1 and c_2 will be in conflict if one contains a generation, and if the other has an occurrence in a brush set at t. We have already seen that we need to keep track of whether a cohabitation class has a generation. The problem is in determining if a cohabitation class had a brush occurrence at a given compilation. To aid in this determination, we propose that each cohabitation class be given a field which points to the node in the history tree corresponding to the last compilation in which some occurrence of the class was *in* a brush set, called the *last brush time*. The extra space required for this is only linear. For cohabitation classes formed in intra-region compilation this field is some reserved value, say NIL, meaning that no element of the class has yet been in a brush set. The same value is used in inter-region compilation if a cohabitation class is formed none of whose elements is in a brush set. Otherwise such classes receive a value equal to the formation time. Note that the determination of the last brush time, as well as the updating of this field of the proper set of classes, can be done at no increase in asymptotic cost, since each of these classes has to be examined in the process of inconsistency detection described earlier. If either c_1 or c_2 has a last brush time of exactly t, we know that it has a brush occurrence at t, so that if the other of c_1 or c_2 has a generation, c_1 and c_2 are definitely in conflict.

Note that a brush time of a cohabitation class, if non-NIL, must be at or after the formation time—on the history tree, it is at or above the formation time. As a by-product of the least common ancestor algorithm, it is easy to tell if this last brush time is before t, i.e., below. If so, no occurrence of it was brushed at t; using this, we can often quickly determine that c_1 and c_2 are definitely not in conflict. Thus, the

only situation in which we cannot decide is when c_1 has a brush time after t and c_2 has a generation, or vice versa, or both. This is one of the reasons that brush sets are recorded in the history tree.

We use these recorded brush sets as follows. As a by-product of the least common ancestor algorithm, we can determine which of c_1 and c_2 are in the left and right subregions and so can tell which of the two brush sets we want to search for some occurrence being in c_1. If we find one, c_1 and c_2 are in conflict; if not, the only possibility is that c_1 had a generation, and c_2 had an occurrence which was brushed. This possibility might have already been eliminated; if not, we use the same technique to see if c_2 had a brush occurrence in this compilation. This finally decides the question of conflict when $ft_1 \neq t \neq ft_2$.

We consider next the question of conflict when exactly one of the formation times is the compilation time, say $ft_1 = t \neq ft_2$. If c_1 was formed by an intra-region union, then we use its decomposition into smaller classes c_{11} and c_{12}, and the fact that c_1 conflicts with c_2 iff c_{11} conflicts with c_2 or c_{12} conflicts with c_2. (Note that in these subproblems, the least common ancestors can be no higher than t, a fact useful in computing them.) Thus, we turn our attention to the case when c_1 was formed during inter-region compilation, and our first task is to determine if conflict between the classes was created at this time. To do this, we again use the known decomposition of c_1 into c_{11} and c_{12}, choosing the numbering so that c_{12} lies in the same subregion as c_2 and c_{11} in the other subregion. To determine if a conflict arises at the compilation t, we apply the "$ft_1 \neq t \neq ft_2$" technique to c_{11} and c_2. This applies because the least common ancestor of c_{11} and c_2 is surely t, and the formation time of neither is equal to t. If there is a conflict here, then c_1 and c_2 are in conflict. If not, the question is decided by whether there is conflict between c_{12} and c_2. (Note that in this sub-problem, the least common ancestor is no higher than the immediate descendant of t in which c_2 (and c_{12}) lies.) This disposes of the case $ft_1 = t \neq ft_2$.

Last, we examine the case $ft_1 = t = ft_2$. (The equality of ft_1 and ft_2 should probably be checked before actually doing the lca algorithm, since this case may arise fairly often.) If t is a kernel region, we determine the class conflict of c_1 and c_2 by enumerating their elements and looking at kernel conflict. The small numbers involved make this quite reasonable. Henceforward, we assume that t is not a kernel

region. If either of c_1 or c_2 arose from an intra-region union, we treat it as c_1 was treated in the same circumstance of the previous case. Thus we consider only the case in which c_1 and c_2 were both formed in an inter-region union at time t. This time we use the decomposition c_i into c_{i1} and c_{i2}, where c_{11} and c_{21} both lie in the same subregion, and c_{12} and c_{22} lie in the other. To determine if a conflict was created at t, we treat the pairs c_{11}, c_{22} and c_{12}, c_{21} by the analysis described in the "$ft_1 \neq t \neq ft_2$" case. If neither results in known conflict, we examine the pairs c_{11},c_{21} and c_{12},c_{22} recursively (knowing in each case that the least common ancestor is lower than t), to see if conflict existed previously. This decides the question of conflict when $ft_1 = ft_2$, the last case to be considered.

The algorithm for class conflict just described could, in theory, be quite expensive. The guess and hope is that in actual programs, it will be pragmatic, because it seems that one would arrive fairly quickly at the case where $ft_1 \neq t \neq ft_2$, which involves no recursion, and for which the heuristic using the last brush time will usually yield a speedy answer one way or the other. If one is willing to pay with space to buy time, a possible improvement is to turn the last brush time field of a cohabitation class into a "brush time list," so that the determination of whether any occurrence of a class is brushed at a given time can be answered more quickly.

An interesting feature of this design is that we do not require an oracle for conflict. Instead, we are relying on solutions to the following as yet undiscussed problems:

Record a brush set.

Enumerate the elements of a recorded brush set.

These are discussed in the next section.

6.7 Representation of Boundary and Brush Sets

Sections 6.4 and 6.6 have reduced the problem of inconsistency detection to the problem of enumerating elements of boundary and brush sets, and of being able to record brush sets. The solutions to these problems involve a common data structure, due to the fact that boundary sets grow by the adjunction of brush sets, and brush sets are formed by boundary sets, minus matched occurrences. A naive representation of these objects would result in a crippling space requirement.

There are two ways in which a naive implementation results in non-linear expense. One is in the construction of a brush set, and the other is in the propagation of this brush set to all of the boundary nodes of the region. Let us work on the latter problem first, assuming that we have a brush set in hand. The goal is to form the (known to be disjoint) union of this brush set with all of the boundary sets, for every boundary node of the region, *in constant time*. What we must have then, is a common place to put the brush set, reachable from all of the boundary sets of a region. To achieve this, we note that:

> A boundary set may be represented as a list (meaning the union) of brush sets, in the order in which the brush sets are added to the boundary set.
>
> Different boundary sets may have a very long common tail, thereby saving much space.

Behold, we have already described just such a data structure: the history tree, with its brush sets! To see how this data structure is used for this purpose, suppose we are given a node for which we desire an enumeration of the elements of its supply set. We first enumerate all of the elements which were in the supply set of the kernel region which the node was compiled into—these can be done economically simply by examining the code for the kernel region. Next, move up to the immediate ancestor in the history tree. Since the history tree also points downward, we can determine whether the node is the left or right son, and thus, we know which of the two brush sets was adjoined to the boundary set of the node during its first involvement in inter-region compilation. The elements of the appropriate brush set can be enumerated. We then advance from the present history node to its ancestor, again enumerate the appropriate brush set, and so on up the tree. This technique ensures linear overall requirements for the space used by boundary sets, and for the time required for their enumeration, up to linear space and enumeration time requirements for brush sets (and modulo null brush sets, which are probably very rare). It is delightful that the history tree plays a crucial role in two seemingly unrelated oracles: class conflict determination, and boundary set enumeration.

We next turn to the problem of brush sets. Unlike boundary sets, these are not constructed as unions of other sets; rather they consist of a boundary set with some specified set of occurrences removed because of matching. I have not been able to devise a technique to have both linear space and linear enumeration time bounds. However, there are techniques to achieve either at the expense of the other, and ways

to combine the techniques and dynamically decide upon the best technique. We first examine a linear space technique. The idea here is to utilize the fact that a brush set can be formed by two other objects: the "base" boundary set from which occurrences are stripped, and the boundary set that provides the matched occurrences. If we can recover these objects, then the elements of the brush set can be enumerated.

The history tree, as with boundary sets, is the key to an efficient representation. Recall that brush sets are "recorded" in each node of this tree; all that is needed to achieve this is a pointer to the arc whose compilation is being remembered. From the arc, we can get to the nodes in question, from which we may enumerate the boundary sets by tracing up the history tree, recursively enumerating other brush sets, until we reach the history node whose brush set is desired.

It is clear that this scheme requires linear space. The rub is that the enumeration of a brush set may require time not necessarily proportional to the number of its elements. With the present representation, we have to enumerate the elements of the base boundary set, and then suppress in the enumeration of the brush set any occurrences matched in the opposing occurrence set. The first problem is that determining whether an occurrence is matched requires in this scheme an enumeration of the opposing occurrence set, which may be expensive. The second is that if too many elements are excluded, we spend a lot of time enumerating not very many elements of the brush set. To solve the first problem, we note that the matched occurrences were detected in the process of generating cohabitation arcs. Since this set must be computed anyway, and about the time brush sets are formed, we have several options. The first is that at any history node, we may keep a list of the cohabitation arcs that were formed because of the compilation of its arc. Then, to exclude matched occurrences, we merely search the list of these arcs, which is presumably much smaller than the opposing boundary set. Note that although this scheme uses extra space, it is not an asymptotic increase, because it requires only one more pointer per history node, and one more pointer per cohabitation arc.

A related method to detect matched occurrences is to utilize the fact that a match between occurrences involves a single variable. This set of variables may be stored in some data structure which allows rapid determination of the question: "is this

variable in the set of matched occurrences". Such a data structure would be a hash table, or a sorted list which could be searched by binary division. The size of these sets is proportional to the number of cohabitation arcs established so again there is no asymptotic increase in space requirements. This or the previous technique is useful if the set of eliminated occurrences is small but the base boundary set is large.

The techniques described above eliminate the necessity of having to enumerate two boundary sets in order to form one brush set. The problem remains that of the perhaps many elements enumerated from the base propagation set, all but a few are matched. This situation can be noted while doing the compilation of the arc, and if too many elements of the base boundary set are matched, a direct representation of the brush set, say as a list of occurrences, may be recorded in the history node.

In each of the above cases, we are expanding space requirements in the hopes of considerable improvement in enumeration time of brush sets. Precise definitions of "large" and "small" will have to wait until we see the performance characteristics of the compiler on real programs.

This and the previous three sections have all been concerned with the problem of inconsistency detection, with the main difficulties arising from the conflict relation. We have taken the approach of being very careful to conserve space, and the history tree has been the key to this, since it is used in three separate but related ways. Although prediction of the behavior of a large program is difficult, it would appear that the representation of conflict and algorithms over it are the most worrisome areas of the efficiency of this compiler. But having the flexibility that detailed conflict information allows is one of the keys to this optimization strategy.

7. Inconsistency Resolution

7.1 Overview

We saw in the previous chapter how to detect an inconsistency. Here we shall study the problem of modifying the code so that it is once again consistent. The means of doing this is usually the insertion of one or more instructions in the code. Such a move may replace what was previously a CHB (cohabit) pseudo-op: we may change CHB V_1, W_1 to MOVE V_1^*, W_1. This corresponds to breaking a cohabitation arc due to an assignment. We may also break a cohabitation arc that corresponds to matching two occurrences $V_1 \rightarrow V_2$. In this case, the move has the form MOVE V_2^*, V_1.

In this work we shall discuss only those resolution techniques that rely on moving data, whether by actual insertion of move instructions or by modifying other instructions so that they do moves implicitly. Other techniques of inconsistency resolution are available. For example, one might rearrange arguments of an instruction: changing ADD V_1^*, W_1 to ADD W_1^*, V_1 may be a useful way to resolve an inconsistency. Other modifications in this class include reordering code to reduce conflict, and duplication of code (especially small subroutines) to reduce cohabitation.

Whenever the code is modified, in particular by insertion of a move instruction, the idea is to make all the data structures appear as if the code had been in that form all along. Thus, a move instruction becomes a kernel region (or part of one), and typically its source is a last use, while its destination is a generation and a first use. We then look at all of the invariants concerning the compiler's data strucure, and make sure they remain true, modifying the data structure if necessary. This would include perhaps adding new split and merge occurrences, forming new cohabitations with any new occurrences added, listing these from the proper point in the history if this is being done, updating cohabitation classes to have the proper set of members and to have the proper "has generation" flag—remember, a move may add a generation. Note that conflict is represented through the history tree, so that the updating of conflict information is almost automatic (beware "last brush time"). In fact, this ease of updating is one of the many advantages in the history tree implementation of conflict. Any other representation seems quite awkward to revise

during resolution.

A further issue in the placement of move instructions is that it may occasionally be desirable to place a move outside the region in which the inconsistency has occurred. What we do in this case is to create a new kernel region or place the move instruction in an existing region different from the one which has caused the inconsistency. This causes no difficulty—eventually, everything is pieced together.

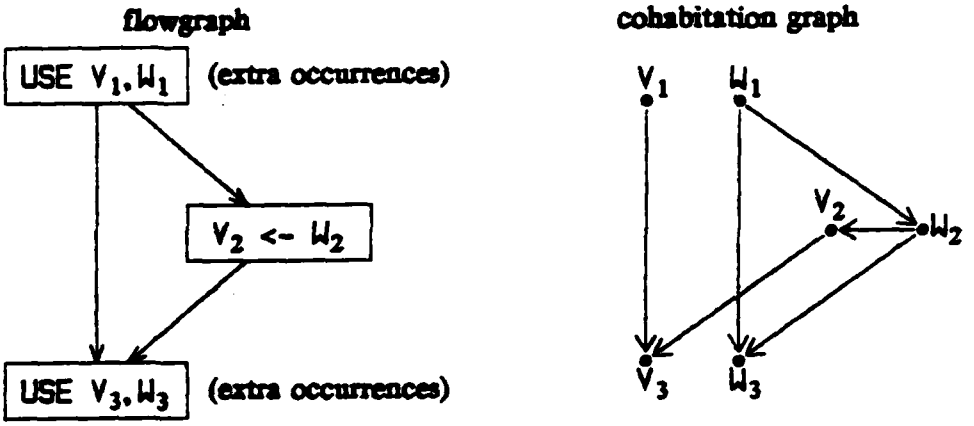
In summary, breaking a cohabitation arc means modifying the program, typically by adding move instructions, and then insuring that all of the invariants regarding extra occurrences, cohabitation, and conflict are correct. Thus, compilation of any arc is oblivious to whether there was previous inconsistency resolution.

7.2 Difficulties

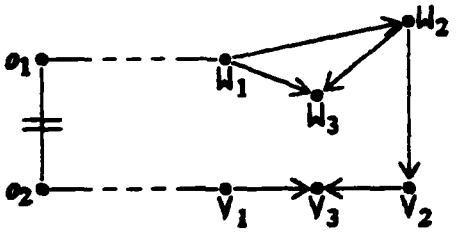
The above sketch gives us faith that if we can only decide where to place move instructions, we will be able to do so without greatly disrupting the compilation strategy proposed in earlier chapters. The real problem is going to be in choosing where to place the move instructions, and how well we do this governs the quality of the code we generate.

One of the most difficult facets of the resolution problem is that mere knowledge of the inconsistent cohabitation and conflict relations does not necessarily tell us how to resolve. Consider the following (wrong) approach. View all the desired cohabitation arcs as in place, so that we have a cohabitation class with internal conflict. Then, remove some subset of arcs, breaking the large class into two or more pieces, each of which is internally free of conflict. Modify the code by placing move instructions between the occurrences of either end of a removed cohabitation arc.

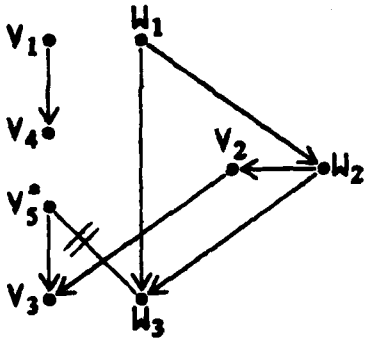
The reason that this approach does not work is that move instructions introduce new generations into the program, those new generations introduce new conflict, and part of this new conflict can in fact be internal to the smaller pieces of the original cohabitation class, pieces which were hoped to be free of internal conflict. As an example of what happens, we consider what happens in a conditional assignment of W to V.



Suppose that V_1 and W_1 are in class conflict, i.e., there are occurrences o_1 and o_2 which are in conflict, and paths from these occurrences to V_1 and W_1.



If we look only at the above relations, it would seem that we can resolve the inconsistency by breaking the cohabitation $V_1 \rightarrow V_3$. However, looking at the program, this is ridiculous on the face of it. Further, if we actually install the move, we have the relations



The conflict between V_5^* and W_3 is because V_5^* is the destination of the move, and is thus a generation. The conflict might also have been between V_5^* and W_1. In either case, the inconsistency remains, via the path:

$$V_5^* \rightarrow V_3 \leftarrow V_2 \leftarrow W_2 \rightarrow W_3 \quad (\text{or } \leftarrow W_1)$$

On the other hand, choosing other arcs to break results in a perfectly acceptable fix

to the inconsistency. For example, breaking $W_2 \to V_2$ obviously will work, since it corresponds to a literal implementation of the assignment.

It is clear that in order to resolve inconsistencies, we must develop a technique which is able to anticipate the effect of the extra generations that are added as a consequence of inserting moves.

7.3 Refinement of Partitions

The first step in resolving an inconsistency is to obtain an object which relates the various occurrences in a class component to their relative appearance in the flowgraph (assume desired cohabitation arcs are in place, throughout this chapter). For example, in the example in the previous section, it is important that V_1 and W_1 are in the same node, as are V_3 and W_3, etc. Another way to see why this is important is to consider what it means to have an occurrence of one variable, say X_1, in a minimal intermediate subgraph of another variable, say $G(Y_1,Y_2)$. If we decide to break $Y_1 \to Y_2$, it may make considerable difference whether the move is placed "above" or "below" X_1. It is this observation which leads to the idea of the common refinement of the intermediate subgraphs of a class component. To introduce the construction of this section, we first consider some relations of equivalence relations.

Definition. Let $\sim_1$ and $\sim_2$ be equivalence relations over a set S. We say that $\sim_1$ is a *refinement* of $\sim_2$ when

$$a \sim_1 b \Rightarrow a \sim_2 b, \text{ for all } a,b \in S$$

An equivalence relation $\sim$ is a *common refinement* of $\sim_i$, $i = 1,2,...$ when it is a refinement of each, and a *coarsest common refinement* $\sim$ has the additional property that any common refinement of each $\sim_i$ is also a refinement of $\sim$.

□

It is a simple matter to show that a coarsest common refinement exists and is unique. In fact, it is given by:

$$a \sim b \Leftrightarrow a \sim_i b \text{ for all } i$$

We now relate refinement of arbitrary equivalence relations to the problem at hand. We first need a notation for the part of a cohabitation graph concerned with a particular variable.

Definition. Let H be any cohabitation graph constructed during compilation (we specifically allow H to have internal conflict due to presence of desired cohabitations). Let V be the variable of some occurrence of H. The *restriction of H to* V, H_V is the set of all occurrences (nodes) of H whose variable is V, together with all arcs of H connecting two such occurrences. (Note that H_V may be disconnected.)

□

Because of the way that extra occurrences are added, any H_V subtends a certain subgraph of the flow graph, and induces a certain partitioning of this subgraph, namely that given by Theorem 4.1. We now characterize the relevant properties of the relationship between H_V and the flowgraph by abstracting the V-free terminology. Let H be a graph (think H_V), and let ν be a map from the nodes of H to the flowgraph (think of the function that takes an occurrence to the node in which it appears). Then much of the terminology that has to do with a variable V can be restated in terms of the "image of H *under* ν", which we abbreviate $\nu(H)$. The following definitions are the major ones of interest. When H is in fact the cohabitation graph H_V, these definitions correspond closely with earlier ones, where "V" is changed to "H". The major difference is that it is H which controls the liveness, and not the ordinary flow rules—we are interested only in the part of the program "seen" by a particular cohabitation graph.

Definition An *H-free path* (in the flowgraph) is one which has no element of $\nu(H)$, except possibly for the first and last nodes. If $p \to q$ in H, the *H-free subgraph* (of the arc or pair of nodes), denoted $G(p,q)$, is the set of all H-free paths from $\nu(p)$ to $\nu(q)$. (The notation "$G(p,q)$" suppresses any mention of H and ν, but this is clear from context, since p and q must be nodes of H and ν is fixed.)

We say that H is *live* at an arc or a node if that arc or node is in some H-free subgraph. Finally, H is a *merge-split partition* (or *ms-partition*) when:

All of its H-free subgraphs have arcs.

H-free subgraphs of distinct arcs of H intersect only when the arcs touch a common node n, and then intersect only at N.

If H is live at a non H-node N, it is live at all arcs touching N.

(The "partition" is of the arcs of the live region of H, not of the entire flowgraph.)

□

The following result is the analogue to Lemmas 4.1 and 4.2.

Lemma 7.1 Let H be an ms-partition, and let $p \to q$ in H. Then P dominates and Q back-dominates all arcs and internal nodes of $G(p,q)$.

Proof. Also analogous to Lemmas 4.1 and 4.2. □

The purpose of all this machinery is not merely to duplicate what we have done for variables. Given a cohabitation graph H involving several variables, we want a single graph H_* which somehow works for all the variables in H "at once". To do this, we need the follwoing notion.

Definition. Let H_1, H_2 be ms-partitions of the flowgraph. We say that H_1 is a *refinement* of H_2 when:

The live region of H_1 includes the live region of H_2.

If $p \to q$ in H_1, and H_2 is live at some arc of $G_1(p,q)$, then all of $G_1(p,q)$ lies in the same H_2-free subgraph.

□

The relation "is a refinement of" is a quasi-ordering (lacking only anti-symmetry), analogous to the identically named relation of partitions. We can define "common refinement" and coarsest common refinement (hereafter abbreviated *ccr*) in the same way. Standard lattice theory results yield the uniqueness of coarsest common refinements (up to equivalence in the quasi-ordering), and reduce the n-ary existence question to one of binary existence. Before giving the details, we provide a rough picture of a ccr of H_1 and H_2. The live region of H is the union of the live regions of H_1 and H_2. The nodes of H are the union of the nodes of H_1 and H_2, together with "H-merge" and "H-split" nodes which have to be added. The arcs of H are obtained in a natural way from those of H_1 and H_2. We begin the formal construction of a ccr with the following algorithm.

Algorithm 7.1 Completed node set of H_1, H_2.

Initially, the completed node set contains $\nu_1(H_1) \cup \nu_2(H_2)$.

If N_1 and N_2 are in the node set and if there are two forward or two backward paths intersecting only at the node N, where the paths are H_1-free and H_2-free and lie in the union of the live regions of H_1 and H_2, then adjoin N to the completed node set.

This corresponds exactly to the construction for extra occurrences at V-merge and V-split occurrences (see section 4.2). We now characterize the ms-partitions we are interested in.

Definition. Let H_1 and H_2 be ms-partitions. A graph H and a map ν is a *common ms-partition* of H_1 and H_2 when:

> $\nu(H)$ is the completed node set of H_1 and H_2.
>
> If n_1 and n_2 are nodes of H, there is an arc $n_1 \to n_2$ precisely when there is an H-free path from $\nu(n_1)$ to $\nu(n_2)$ where H_1 or H_2 is live at every arc on the path.

□

Theorem 7.1 Let H_1 and H_2 be ms-partitions. A common ms-partition of H_1 and H_2 is an ms-partition and is a ccr of H_1 and H_2. Conversely, any ccr of H_1 and H_2 is a common ms-partition.

Proof. That H is an ms-partition is essentially shown in Theorem 4.3, because the construction of a common node set for H is exactly what was done when adding extra occurrences of the variable V. Note that Lemma 7.1 plays the role of Lemmas 4.1 and 4.2.

It is also not difficult to show that H is a refinement of H_1. Let H_i be live on the arc A, so that A is in $G(p,q)$ where $p \to q$ in H_i. Let N_j be the last node in the image H appearing in this path before A, and let N_k be the first such node after A. Since this path lies entirely in $G(p,q)$, H_i is live at every arc of it, so that by definition of common ms-partition, $n_j \to n_k$ in H, so that H is live at A. Thus, the live region of H includes the live region of H_i, the first requirement of refinement. The second requirement is the content of Corollary 4.3, so that H is indeed a common refinement of H_i.

What remains is to show that H is as coarse as possible. Let H_0 be a common refinement of the H_i. Since its live region must contain the union of the live region of H_i, which is exactly the live region of H, proving the first property required of sharing that H_0 is a refinement of H. Let $p \to q$ in H_0. We claim that $G(p,q)$ has no internal nodes in the common node set. Thus, if H is live anywhere in it, all of $G(p,q)$ lies in the same H-free subgraph, essentially by Corollary 4.3. We use an inductive proof, following the inductive construction of the common node set. To start the induction observe that $G(p,q)$ is H-free, since H_0 is a refinement of H_1 and H_2. Assuming the truth of the claim inductively, consider paths from nodes N_1 and N_2 in the node set to a node N internal to $G(p,q)$. Since N_1 and N_2 are outside

$G(p,q)$, the paths must go through p (if they are forward) or through q (if they are backward); this by Lemma 1 and the fact that H_0 is an ms-partition. The paths thus intersect at a point other than N, so that N cannot be added to the common node set at this point. This completes the proof of the claim, and thus of the Theorem.

□

A few remarks are in order concerning the relationship of this result and the extra occurrences discussed in Chapter 4. The initial motivation given for extra occurrences was to aid in finding the proper place to put moves, and the V-merge and V-split nodes might have seemed to be an ad hoc construction. We can now see that they arise in a completely natural way. The notion of a V-free subgraph is an almost inevitable way to formalize the "data flow" of a variable from one use to a next use. The definition of ms-partition formalizes the idea of partitioning the live region of a variable using V-free subgraphs, and the definition of refinement of ms-partitions generalizes the usual notion of refinement to the situation in which the domains of the relation may overlap, but are not necessarily equal. If we take each V-free subgraph individually, and consider all other occurrences of V to be isolated, we have an ms-partition whose live region is just that *one* V-free subgraph. If we want to divide up the entire live region of V, the coarsest common refinement of all such ms-partitions is the only mathematically reasonable thing to do. This forces on us the V-merge node and V-split node construction of Chapter 4.

In this chapter, our motivations are of course different. Here we want to relate the partitions of the live regions of several variables to common parts of the flowgraph. We can now define H_* to achieve the effect we wanted.

Definition Given a cohabitation graph H, the *refinement* H_* of H is defined to be the coarsest common refinement of H_V, where V ranges over all the variables with occurrences in H.

□

Note that the H_*-free subgraphs are regions in which we have free choice regarding the placement of moves involving variables of occurrences of cohabitation graph H. The problem of deciding where to place moves factors into two problems—a certain optimization problem on H_* which will yield a choice of which variables to move on

which arcs, and then a code modification problem—given that a variable is to be moved on an arc of H_e how do we best place code to do it in the H_e-free subgraph subtended by the arc.

7.4 Maximal Cohabitation Classes

Conflict often arises as a global phenomenon—a variable must retain its value from an occurrence V_1 to another occurrence V_2, but there is an intervening generation W_1 of another variable. Thus far our representation of conflict has retained this non-local feel. The refinement H_e of a cohabitation class makes it possible to represent conflict so that *all* conflict looks like kernel conflict. This is a first step in seeing how to resolve an inconsistency of H.

Suppose that to each node of H_e we attach an occurrence of all the variables of H which are live at that point, where we use the occurrences already on the line of code, if there are any, and make new ones for other variables. Extending the rule for kernel conflict (page 33) to these new occurrences, any conflict within H will appear as kernel conflict, by the construction of H_e. This kernel conflict is the seed of what we call *local conflict*, where the term is chosen because it can be seen by looking only at the node in question.

In discussing the techniques of this chapter, it is convenient to imagine that whenever $n_1 \to n_2$ in H_e each non-last occurrence of n_1 has a cohabitation arc to some non-first occurrence of n_2, where this cohabitation arc always connects occurrences of the same variable. The usual situation is that a variable will have only one occurrence among non-first or non-last uses, so that the arc is redundant. The first application of these (perhaps imaginary) cohabitation arcs is in the following result, which tells how local conflict arises.

Lemma 7.2 Let $n_1 \to n_2$ in H_e and suppose we have occurrences V_i, W_i at n_i, $i = 1$ and 2, where V_1 and W_1 connect (respectively) to V_2 and W_2. If it is somehow known that V_1 and W_1 cannot be in the same cohabitation class, we can also conclude that V_2 and W_2 cannot be in the same cohabitation class, if the only change to the program is the insertion of move instructions.

Proof. Suppose V_2 and W_2 are in the same cohabitation class, but that V_1 and W_1 are not. Then $G(n_1,n_2)$, which had no occurrences of V or W at the time of the

construction of the cohabitation class, must have been modified with one or more instructions which moved one or both of V and W into a common place. This changes the values of one of the variables, violating program semantics. In terms of cohabitation and conflict, the destination of the instruction(s) must have been a generation of one of the variables, and so is in conflict with the other variable. But if V_2,W_2 cohabit, this is again an inconsistency.

□

Unlike many of the results and techniques of this paper, this Lemma is decidedly non-dual. Formally, this might be viewed as being a consequence of the fact that in a move instruction, the source, i.e., last use, is not a generation, but the destination, i.e., first use, is a generation. At a more philosophical level, the non-duality arises because entropy always increases; in programs, this happens when a memory location is clobbered.

Using the seeds of local conflict and Lemma 7.2, we can "grow" local conflict. At each node of H_* the algorithm below partitions its occurrences into what we call *maximal cohabitation classes*, or *mxcc's*. These are maximal in the sense that no matter how move instructions are inserted to remove inconsistencies, the final cohabitation classes, restricted to any node, will be contained in a maximal cohabitation class at that node. Initially, each generation is in a mxcc whose only other occurrences are in kernel cohabitation with it. All other occurrences are placed in a single mxcc. Then for any arc $n_1 \rightarrow n_2$ of H_* we can obtain a new mxcc-set at n_2 from the one at n_1 as follows:

Algorithm 7.2 Grow a mxcc-set

GR0 Initialize the derived mxcc-set to be the mxcc-set of n_1.

GR1 If an occurrence dies out along $n_1 \rightarrow n_2$, remove it from its mxcc.

GR2 Replace each occurrence in the derived mxcc-set by the one of n_2 to which it is connected by the cohabitation arc along $n_1 \rightarrow n_2$.

GR3 The only occurrences of n_2 not presently in the derived mxcc-set are first uses at n_2. If one of these occurrences "kernel cohabits" with some non-first use, put it in the same derived mxcc as the one with which it cohabits (this always applies if the first use is not a generation, in which case it is o_1 in CHB o_1,o_2). The remaining occurrences, all generations, are put in mxcc's according to their kernel cohabitation relation.

GR4 Replace the mxcc-set at n_2 by the coarsest common refinement (of simple partitions) of the current mxcc-set at n_2, and the derived mxcc set.

This algorithm applies only to one arc, and by itself is not an algorithm for getting mxcc-sets everythere. However, it fits into the general class of "weak interpreter" techniques, the theory of which guarantees a "strong as possible" global set of mxcc-sets that is consistent with the seeds and the growth rule. To calculate this global set, we just continue applying the above rule till things settle down. Algorithmic aspects will not be discussed further here. We do, however, prove that such a global set of mxcc-sets tells us what we want to know about conflict.

Theorem 7.2 Suppose we label the nodes of H_s with mxcc sets, beginning with the seed mxcc-sets, and then by repeatedly growing new mxcc-sets. Suppose we eliminate inconsistencies by the insertion of move instructions. Then:

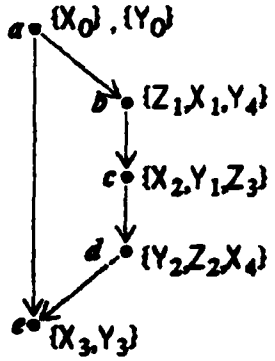
> If o_1 and o_2 are both non-last occurrences at a node of H_s and are in different mxcc's before the modification, then they will not cohabit after the modification

Proof. This is true of seed mxcc-sets because at least one of the generations will be or will kernel cohabit with a generation, and by assumption, the other will not be a last use. Thus, they will be in kernel conflict. Lemma 7.2 says that the same property will hold of the derived mxcc-set constructed in GR1-GR3.

All that remains is to show that if the result holds for two mxcc-sets at a node of H_s, it holds at their coarsest common refinement. Let o_1 and o_2 be in different mxcc sets. By the remark in the previous section about common partitions of sets, we know that o_1 and o_2 are in different mxcc's in at least one of the two mxcc-sets. That mxcc-set tells us that o_1 and o_2 cannot cohabit in a program changed only by the addition of move instructions. This proves the desired property of the coarsest common refinement.

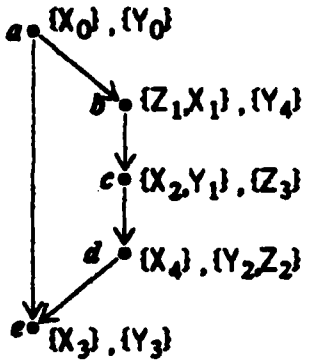
Incidentally, we can also observe that no information is lost in this step, i.e., the conflict of both mxcc-sets is reflected in coarsest common refinement. Suppose o_1 and o_2 are in the same mxcc's of the coarsest common refinement. Then they are in the same mxcc's in both of the original mxcc-sets, so no stronger statement was known previous to the replacement of the current mxcc-set by the new one. □

We now give an example of how this works. Return to the conditional exchange example of section 3.4, page 16. We first give H_s with the initial mxcc-sets:

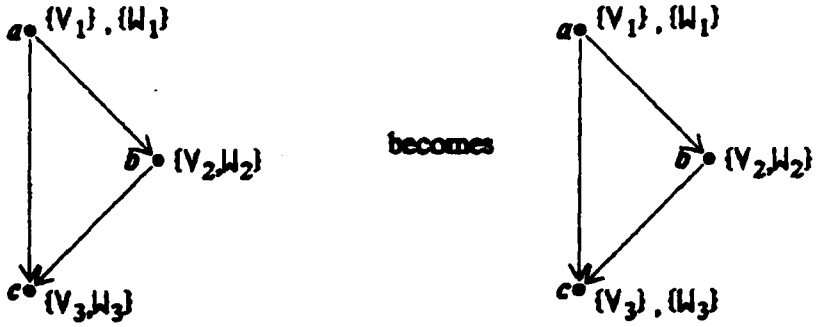


X_4,Y_4 and Z_3 are occurrences added because the variables are live at their respective nodes

Note that at point *a*, we are assuming that we somehow know that X_0 cannot cohabit with Y_0. Suppose we derive a mxcc-set along *a*→*b*. By GR3, the derived mxcc-set will be $\{Z_1,X_1\},\{Y_4\}$ which is the coarsest common refinement of itself with the current mxcc-set at *b*, and so replaces it, by GR4. Then consider *b*→*c*. At the end of GR1, the derived mxcc-set is $\{Z_1\},\{Y_4\}$, while at the end of GR3, it has become $\{X_2,Y_1\},\{Z_3\}$, which becomes the mxcc-set at *c*. Next, consider *c*→*d*. We see that the new mxcc-set for *d* is $\{X_4\},\{Y_2,Z_2\}$. When we process *d*→*e*, the mxcc-set for *e* becomes $\{X_3\},\{Y_3\}$. Finally, consider *a*→*e*. The derived mxcc-set is $\{X_3\},\{Y_3\}$ which is the current mxcc-set. Thus, no change occurs, and we have obtained a global assignment of mxcc-sets.



As another example, we look at conditional assignment from section 7.2, page 59. We assume that the seed of local conflict is node *a*.



Here the only change was to the mxcc-set on node *c*. For node *b*, V_2 is a first use, and V_1 dies out along *a*→*b*.

7.5 Splits and Twists

Suppose we have H_o labeled with mxcc's. We wish to modify this labeling in a way which reflects the insertion of move instructions, so that we are finally able to assign the mxcc's to cohabitation classes consistent with the generated code. We begin by finding some condition on the mxcc's which makes this trivial. This condition is most conveniently discussed in terms of the following object, which we do not propose actually implementing.

Definition. The *mxcc-graph* has nodes which are mxcc's, and arcs induced from the cohabitation relations on the elements of the mxcc's.

□

In terms of this graph, the consistency condition is that there not be an undirected path in it between distinct mxcc's at the same node of H_o. We will examine inconsistencies by looking at the image in H_o of the mxcc-path from the inconsistency. The inconsistency condition is easier to work with when it is broken down into two conditions. The first is one which is can be seen along a single arc of H_o.

Definition. Let $n_1 \rightarrow n_2$ in H_o. Suppose some mxcc of n_1 has arcs to distinct mxcc's of n_2. We say that this mxcc *splits* along $n_1 \rightarrow n_2$.

□

An example of splitting is the conditional assignment example. Refer to the previous figure. The derived mxcc of $\{V_2,W_2\}$ along *b*→*c* is $\{V_3,W_3\}$, which is not a mxcc of node

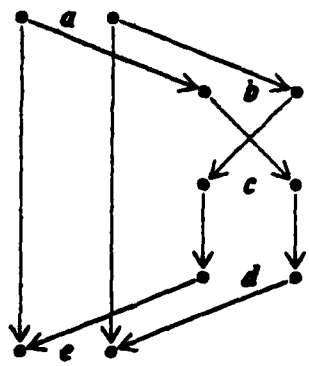
c; instead, we find $\{V_3\}$, $\{W_3\}$.

It is clear from the definition of mxcc-graph that each mxcc-arc "belongs" to a certain arc of H_s. Note that when there is a split, the path between distinct mxcc's at a node belongs to a cycle of arcs in H_s consisting of one arc repeated twice—first backward, then forward. We know that splitting does not cover all inconsistencies, for if we look at the mxcc sets computed for the conditional exchange example (page 69), we see that there is no splitting. But we know that something must be wrong, because there is inconsistency. The problem is captured as follows.

Definition. An undirected path in the mxcc-graph between distinct mxcc's at the same node of H_s is called a *twist* if the cycle of H_s-arcs to which it belongs is simple (has no repeated arcs).

□

We draw the mxcc-graph for the conditional exchange example, which motivates the term twist (mentally fill in mxcc labels in the same order as they were listed in the previous section).

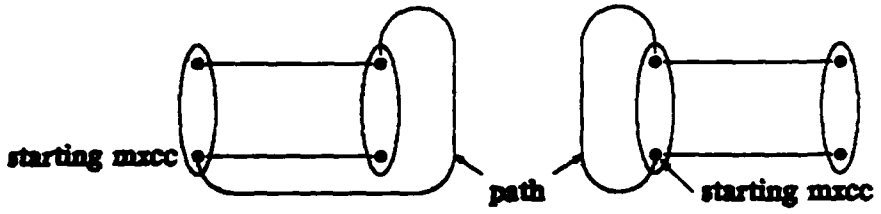


This could be drawn most symmetrically on a Moebius band.

Theorem 7.3 Suppose a mxcc-graph is free of splits and twists. Then there are no inconsistencies (and the components of the mxcc-graph are the desired cohabitation classes).

Proof. Suppose we have an inconsistency, i.e., an undirected path of mxcc-arcs between two mxcc's on the same H_s-node. We may assume that this path is of minimal length. We want to find either a split or a twist. Look at the cycle in H_s to which the undirected path belongs. If the H_s-cycle has no repeated arcs, we have a twist and are done. Suppose some arc in the H_s-cycle is repeated, and examine the

mxcc-arcs belonging to it. If these arcs share a mxcc, they must both leave that mxcc, because distinct arcs can't enter a mxcc after the growing of local conflict which was done in the previous section. Thus, this mxcc is split, and we are done. The only other possibility is that there are four distinct mxcc's touched by these two mxcc-arcs. Start at any one of these four mxcc's, and follow the mxcc path until one of the four is encountered. This mxcc-path cannot come back to the same mxcc or to the mxcc on the other end of the mxcc-arc, or we would have a mxcc-cycle, meaning that the original undirected path did not actually contain all four mxcc's. There are essentially two possibilities, most easily described by their pictures (ellipses enclose mxcc's at a single node of H_o):



In the first case, we may extend the mxcc-path by one more arc, and wind up at a distinct mxcc at the same node of H_o. In the second case, we wind up at such a mxcc just by the path. In both cases, we have contradicted minimality of the length of the mxcc-path.

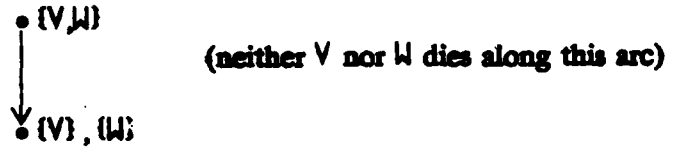
□

The decomposition of the problem of inconsistency resolution into splits and twists sets the stage for the rest of this work. In the next chapter, we will consider the problem of split removal. Given H_o and the information of the accompanying mxcc-graph, the techniques of that chapter will tell how best to insert moves so that if H_o is re-derived, there will be no splits. It seems likely that split-removal will resolve most inconsistencies, although we know that it cannot resolve all of them. Chapter 9 discusses the problem of untwisting. The techniques there assume an H_o and a mxcc-graph that is free of splits, and tell how best to insert moves resulting in consistent cohabitation and conflict relations. Synergistic interactions between split-removal and untwisting are not considered. This question will have to be re-opened it empirical evidence refutes the intuition that little would be gained by such techniques.

8. Split Removal

8.1 The Two Variable Case

We begin our discussion of split removal with the special case in which H has only two variables, neither of which is replicated. In this case, each node of H_s has at most two occurrences, and if there are two, there is either one mxcc or two. A split always has the following form:



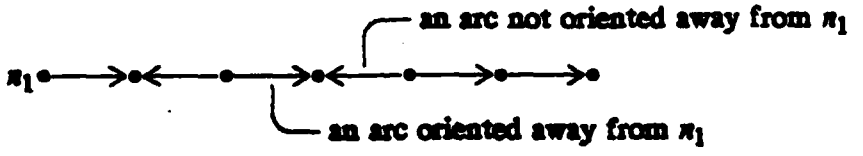
(neither V nor W dies along this arc)

There may be several such splits. Form the *modification subgraph* M of H_s:

Algorithm 8.1 Construct a two variable modification subgraph
Include in M all arcs of H_s along which a split occurs.
for $n \leftarrow$ the top node of each split
Adjoin to M all undirected paths starting at n such that:
(1) both variables are alive along an arc, and
(2) both variables are in the same mxcc at a node.

This subgraph has the following important property.

Theorem 8.1 Let n_1 be a node of M at which there is CHB whose arguments are V and W. Let n_2 be a node of M at which V and W are in different mxcc's, and consider any undirected path betwen n_1 and n_2. Suppose that the code is modified by the insertion of moves (explicit or otherwise) in such a way that there are no remaining splits and no replications. Then there is a move inserted in the H_s-free subgraph subtended by some arc oriented *away* from n_1 on the path. In pictures:



Proof. Let H_s' be the ms-partition computed after the moves are inserted; grow mxcc-sets in H_s'. The proof is based on comparing H_s' to H_s, the original ms-partition. All of the nodes along our undirected path essentially lie in H_s', but the arcs between two nodes may be replaced by a finer subgraph, because of extra occurrences added if a move is inserted internally in $G(n_i,n_j)$, where n_i and n_j are adjacent on the undirected path. Nevertheless, if $n_i \rightarrow n_j$ there will be a path in H_s'

from n_i to n_j, and dually if $n_j \rightarrow n_i$.

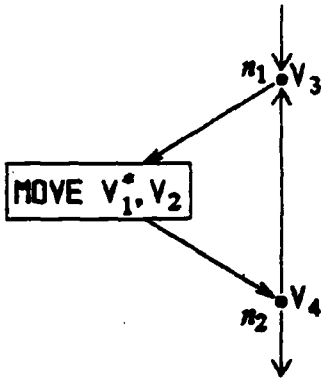
In the undirected path, the first arc must *leave* n_1, not enter, because if n_1 has a CHB, the left operand is not live along any arc entering n_1. Thus, if n_1 has been modified (the CHB changed to a move), the Theorem holds. Otherwise, the mxcc-set of n_1 as a node of H_s' will have only one mxcc, because the CHB is intact, and because there are no replications. Let n_3 be the last node along the undirected path which has only one mxcc in H_s' (we may have $n_3=n_1$), and let n_4 be the next node. Note that $n_4 \rightarrow n_3$ is impossible, since there would be a directed path from n_4 to n_3 in H_s', and by the Growth Rule, if there are two mxcc's at n_4, there must be two at n_3 —remember, both V and W are live through $G(n_4,n_3)$, and insertion of moves does not change this. Thus, we must have $n_3 \rightarrow n_4$, and the move instruction must appear in $G(n_3,n_4)$, as desired. □

The reason that we are interested in this Theorem is that it shapes how we look for code modifications. Consider the conditional assignment example started in section 7.2 and for which we computed mxcc-sets in section 7.4. The only split is along the arc $b \rightarrow c$. If we compute the modification subgraph for this case, we see that it includes only this one arc. Thus, there is essentially no choice in how to resolve the inconsistency: we change the CHB to a move instruction.

A further restriction on where the modifications occur is given in the following result.

Lemma 8.1 *Let M be constructed as above. If moves that are added to the code do not create replications, those moves do not appear in H_s-free subgraphs subtended by arcs in strongly connected components of M.*

Proof. A node containing a CHB has *no* incoming arcs, and is thus not in a strongly connected component (scc). Thus, a move in an scc will be the form MOVE V,V or MOVE W,W. The picture is:



Since V is live on the entry and exit arcs of this scc, n_1 will be a V-merge node and n_2 will be a V-split node. Let V_3 be the merge occurrence of V at N_1, and V_4 be the split occurrence at n_2. We claim that V_1 and V_2 are both not last uses, i.e., that the move replicates V. If V_1 is a last use, the move instruction is superfluous, and would not have resolved an inconsistency. Suppose V_2 is a last use. Then V_1 must cohabit with V_4, since this is the only choice. Since we also have V_3 cohabiting with V_2 and V_4 with V_3 (by $n_2 \rightarrow n_1$), we see that V_1 cohabits with V_2. This is also absurd, since it too would not resolve an inconsistency (in fact, it formally causes one, since V_1 and V_2 are in intra-line conflict). Thus, V_2 is a last use, contradiction. □

It must be noted that it is occasionally useful to create replication in just the above way. However, it is a second-order optimization, and is not treated here.

The previous two results allow us to approximate an optimal solution to the two variable replication-free split removal problem, by converting it to an efficiently solvable graph problem. Assume that along any arc of M not in an scc, the cost of moving V is equal to the cost of moving W. Then we can call this the common cost of the arc of H_M. For arcs which are in scc's, we assign a cost of infinity, meaning that no move is allowed on the arc. Call all of the nodes of M having cohabitations of V and W *sources*, and call nodes of M having two mxcc-sets *sinks*. What we are interested in is:

Definition. A *split-removal modification* (or *srm*) is a set S of arcs having the property that every undirected path from a source to a sink contains an element of S oriented away from the source.

□

We want the srm of minimal cost. This is very close to the maximal-flow-minimum cut problem, the difficulty being that here we are *quantifying over* undirected paths and oriented arcs, whereas the standard max-flow-min-cut works on directed paths and oriented arcs, or undirected paths and unoriented arcs. We can convert our problem to the fully directed case by a simple technical device.

Lemma 8.2 Let M be a graph whose arcs have costs and whose nodes are labelled as sources, sinks, or neither, where all sources and sinks are not in scc's. Obtain M_0 from M in the following way: for every arc $n_1 \rightarrow n_2$ of M, adjoin an arc $n_2 \rightarrow n_1$ with infinite cost. Then the set of srm's of M is equal to the set of finite-cost cut-sets (in the usual flow-theoretic sense) of M_0.

Proof. There is an obvious bijection between the set of undirected paths of M and directed paths of M_0, so that when we consider finite-cost srm's and cut-sets, the inclusion remains true. The only thing remaining to show is that a finite cost cut-set maps to a finite cost srm. This holds because all arcs in M_0 and not in M have infinite cost. Thus a finite cost cut set of M_0 contains only arcs of M; these obviously constitute an srm of M; by the bijection of undirected paths of M and directed paths of M. □

Now that we know how to efficiently compute an optimal srm, we show that it does in fact yield the desired effect on the program. The following is a partial converse of Theorem 8.1.

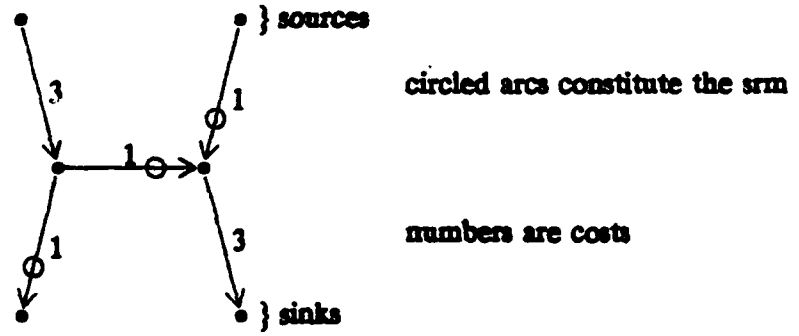
Theorem 8.2 Given an optimal srm, suppose we insert a MOVE V_1^*, V_2 at the point in the flowgraph corresponding to each arc of the srm. Then H_s', constructed as in the proof of Theorem 8.1, is free of splits and replications.

Proof. The crucial part of this proof is to show that every *directed* path from a source to a sink encounters exactly one element of the srm. Suppose we can show this. Then the srm partitions M, and thus the flowgraph, into places where V and W cohabit, i.e., are on a directed path from a CHB without an intervening element of the srm, and those places where V and W are in different mscc-sets, i.e., where there is a directed path to a sink, which is someplace in the unmodified flowgraph were we knew that V and W cannot cohabit. The boundary between these two pieces is exactly where instructions of the form MOVE V_1^*, V_2 are in place, and so this whole arrangement is what is obtained by the growth rule.

To prove the result, we obtain a contradiction from assuming that some forward path in M from a source to a sink encounters two arcs of the srm. We make heavy use of the optimality assumption. First, we introduce the concept: a node n is *source-separated* if any undirected path between a source and n has some arc of the srm, oriented away from the source. Call the dual concept *sink-separated*. Let n_1 be pointed to by an arc A_1 in the srm on a directed path from a source to a sink having two srm arcs, and let A_1 not be the last such arc. We claim that n_1 is not source-separated. If so we claim that A_1 can be dropped from the srm, and the remainder will still be an srm. The only undirected paths that this would affect are those containing A_1 oriented away from the source. But in such paths, since n_1 is source-separated, we know that there is another arc of the srm, properly oriented. Thus, A_1 is unnecessary in the srm, contradicting its optimality. We conclude that n_1 is not sink-separated.

Let A_2 be the next arc in the srm on the forward path after A_1, and let n_2 be the arc which A_2 leaves. We may dually conclude that n_2 is not sink-separated. But then we have an srm-free undirected path from a sink to n_2 (n_2 not sink-separated), and from n_2 to n_1 (because A_2 is the next arc in the srm after A_1, and from n_1 to a source (n_1 is not source separated). But this contradicts the assumption that we were given a srm. Thus, we can conclude that on any directed path from a source to a sink, there is only one arc of an optimal srm. □

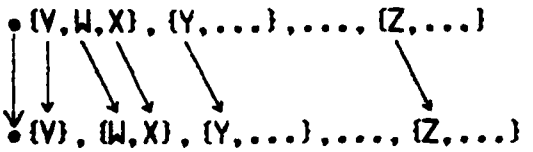
The reader should be aware of the fact that an optimal srm may have several arcs on an undirected path from a source to a sink.



8.2 The Difficulty of Split Removal

In this section we shall formulate the general split removal problem, and show that is is NP-hard. The result is of course not tremendously useful in designing the code generator, other than halting the search for an efficient algorithm. However, the proof is instructive both in showing the source of the combinatorial difficulty, and in suggesting approximation heuristics.

The split removal problem involves both the ms-partition H_s and the associated mxcc-graph. Two nodes and a connecting arc of H_s have associated mxcc's and mxcc-arcs, depicted thus:



(Occurrence subscripts have been omitted.) When we are worrying about split removal, the algorithm for growing mxcc-sets has already been applied. Thus we may see splits, as in the mxcc {V,W,X} above, but we will never see arcs from different mxcc's going into the same mxcc.

We think of split removal as effected by the insertion of a set of moves having the property that after they are inserted and mxccc-sets regrown, there are no splits. The moves have the effect of breaking the cohabitation arc into the destination mxcc-class. In the above case, the destination of the move might be the mxcc-set {W,X} on the lower node. This removes the split, as we can see locally. It is also possible to break up a mxcc well in advance of a split, in which case, it can be seen to remove the split only by growing mxcc-sets.

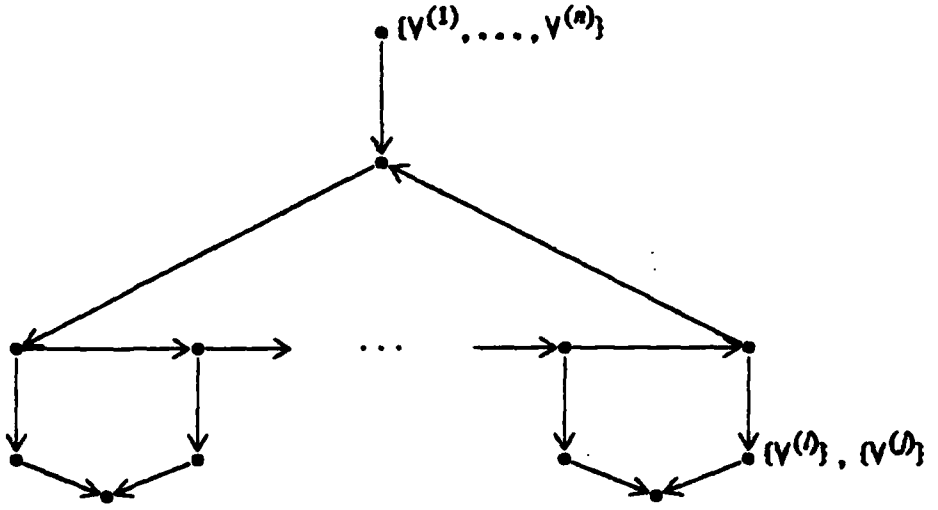
In purely graph-theoretic terms, a split removal can be thought of as a set of sets of cohabitation arcs, each individual set of cohabitation arcs belonging to a common H_s-arc and originating in a common mxcc. In the above example, a set of cohabitation arcs would be the singleton V→V arc, or the set consisting of the W→W and X→X arcs. Each set of cohabitation arcs corresponds to a single move instruction, so that a split removal set is defined to have the property that removing the arcs, partitioning the mxcc-sets at the destinations, and growing, leads to a split-free mxcc-graph. The cost of a split removal might depend in a complicated way on the

costs of the cohabitation arcs involved, but we will prove NP-hardness in the restricted case where the cost depends only upon the H_e-arc.

Definition To *color* a graph is to assign integers (colors) 1, ..., χ to its nodes such that adjacent nodes do not receive the same colors. A *minimal coloring* is one which uses a minimum value of χ. We say that χ is the chromatic number of the graph.

□

The problem of minimally coloring a graph is known to be NP-complete [3]. We shall show how to transform a graph to be colored into a split removal problem, such that the solution of the split removal problem will give a minimal coloring of the graph, thereby proving that the split removal problem is NP-hard. Given a graph, we view each node i as corresponding to variable $V^{(i)}$, $i = 1, ..., n$, where n is the number of nodes. We construct H_e thus:



We have depicted the mxcc-set at the top node; this same mxcc-set is also the mxcc-set at all the nodes in the loop. We have also depicted the mxcc-set on the lower right node. This corresponds to an arc between nodes i and j of the graph to be colored. In fact, for every arc in the graph to be colored, we have *two* exit arcs from the loop, each ending in an H_e-node with mxcc-set like the above, and each node of the pair connected to a common node. Thus, if the graph to be colored has a arcs, the H_e constructed above has $2 \cdot a$ exit arcs from the loop $2 \cdot a$ more arcs after these, $2 \cdot a + 1$ arcs in the loop, and one more arc A, on top. The cost on the arcs in the loop we put at infinity; all other arcs have a cost of 1—this simply means that the frequency is so low that the cost of inserting a move is simply the space for it. It is clear that the desired H_e can be constructed in polynomial time. Note that

growing mxcc-sets would change nothing.

Now, suppose we are given an (optimal) solution to the split removal problem, and that the mxcc-sets have been reinitialized in accordance with the inserted move, and regrown. Let us consider variables $V^{(i)},V^{(j)}$ where (i,j) is an arc in the graph to be colored. Suppose that $V^{(i)}$ and $V^{(j)}$ are still in the same mxcc at the entry node of the loop. Since no moves have been inserted in the loop, there must be moves on each of the two exit arcs for (i,j). But is this is the case, we can improve the solution by removing the two moves and placing a single move on arc A, contradicting the optimality of the solution. We conclude that if (i,j) is an arc of the graph to be colored, $V^{(i)}$ and $V^{(j)}$ are in different mxcc's at the entry node to the loop.

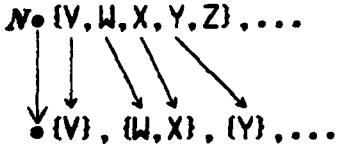
The correspondence with the coloring problem now follows. A solution of the split removal problem minimizes the number of mxcc's at the mxcc set at entry to the loop. We color each node of the graph to be colored by its mxcc; by what we have said, this is a coloring of the graph. Conversely, any coloring of the graph in χ colors can be turned into a split removal with cost $\chi-1$. This proves

Theorem 8.3 The split-removal problem is NP-hard.

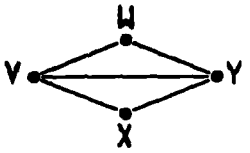
8.3 The Eventually-Separate Relation

We have seen that the two-variable case of split-removal is easy, and that the general case is hard. In later sections of this chapter we shall outline an approximate solution to the difficult case. It will reduce split removal to several max-flow-min-cut problems and several graph coloring problems. Known heuristics may be applied to the graph-coloring problems (see [4]). These represent some of the intrinsic difficulty of the split removal problem. When there are only two variables, the graph coloring problems are easy, and the approximation algorithm reduces to the algorithm proposed earlier, so it is exact.

In this section, we will introduce a relation that is important in constructing both the max-flow-min-cut and the coloring problems. We begin by looking at a split. As our canonical example, we will use



The split induces a relation on the occurrences of a mxcc at *N*, which we may draw as a graph:



We call this relation *eventually-separate* because there is a forward path from *N* that eventually leads to a node where occurrences so related are in separate mxcc's.

Suppose that there is an assignment Y←Z at *N*, which we see as a CHB Y, Z and an intra-line cohabitation from Z to Y. It is clear that Z is eventually-separate from anything from which Y is eventually-separate, and we explicitly include the pairs (V, Z), (W, Z) and (X, Z) in the eventually-separate relation at *N*. This is called *completing* the relation. To summarize what we have said so far:

Algorithm 8.2 Initialize eventually-separate relation.

```
for A ← each arc is H.
    for m ← each mxcc at the beginning of A
      for o1,o2 ← each pair of occurrences in m
       if o1 and o2 map into different mxcc's along A
           make o1,o2 eventually-separate
    Complete the eventually-separate relation of m
```

Observe that non-splits result in null relations. A two-way split produces bipartite complete graphs within a mxcc; it is one of the places in which the assumption of small mxcc's plays a role in practicality. Even with four occurrences in a mxcc, the largest number of eventually-separate pairs is six.

Continuing our above example, let us look at what might occur at a node preceding *N*, and its corresponding eventually-separate relation:

(Evidently X and Y are dead along this arc.) We will modify the eventually-separate

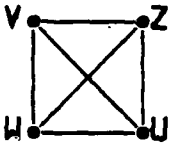
relation at N_1 by pulling back the relation at N and completing it.

Before giving the algorithm for propagating the eventually-separate relation, we discuss an interesting consequence of completing it: it is possible for an occurrence to be eventually-separate from itself. This happens whenever there is an intra-line cohabitation between two occurrences that are eventually-separate. As an example of this, look at the cohabitation graph of the first figure of section 7.2, and its corresponding H_* in the last figure in section 7.4. Since there is an intra-line cohabitation arc from W_2 to V_2, and since V_2 and W_2 are eventually-separate (by initialization), W_2 will be eventually-separate from itself. Thinking of how this would appear in the graph of the relation, we say that W_2 has *self-loop* on it. During split removal, any self-loop will at some point be in a source node of a modification subgraph. We saw this in the conditional assignment example of section 8.1, page 74. The algorithm for propagating the eventually-separate relation is phrased so that self-loops are not propagated.

Algorithm 8.3 Propagate eventually-separate relation along A

```
for m ← each mxcc at the beginning of A
    for o1,o2 ← each pair of distinct occurrences in m
        if o1 and o2 map to eventually-separate occurrences along A
            make o1,o2 eventually separate.
```

Continuing our above example, we would get a new eventually-separate relation at N_1:



Just as we propagated the growth of mxcc-sets forward, we propagate growth of the eventually-separate relation backward, until the relation stabilizes. Along an arc A of H_* from N to N', the eventually-separate relations of several mxcc's of N' may contribute to the same mxcc at N. However, since the mxcc-sets have been "grown" (Algorithm 7.4), a given mxcc of N' can affect at most one mxcc at N. After growing the eventually-separate relation, we will have an eventually-separate relation on each mxcc of each node of H_* with the following property: if all first uses (corresponding to dead variables on the incoming arc) are removed, the relation maps backward along incoming arcs to a sub-relation (think, sub-graph) on a mxcc on a previous node of H_*.

We said in the introduction to this section that graph colorings would enter into the split removal process. The graphs that are colored are—almost—the eventually-separate relations on mxcc's. The colors correspond to the cohabitation classes that will exist once splits are removed. In some situations, colors on occurrences at a node of H_e merely signify the cohabitation classes into which the cohabitation class of an occurrence is eventually copied; in others, differently colored occurrences at a node of H_e will be in different cohabitation classes.

Technically speaking, it is impossible to color a graph with self-loops, so it is not always possible to exactly color the eventually-separate relation. Further, the "coloring" that we can't quite do must be propagated from node to node of H_e, reflecting the cohabitation classes that we are trying to form. This propagation runs into other difficulties. Both sets of difficulties are taken care of as we construct a modification subgraph, (the subject of the next section) analogous to the one we used in the two variable case. In the general case, we will make several such constructions.

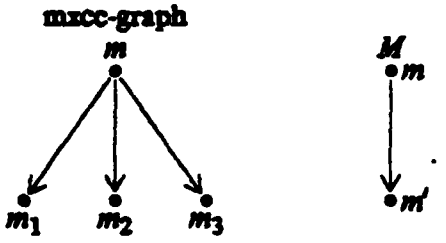
8.4 Construction of a Modification Subgraph

The construction of a modification subgraph is an attempt to get a good approximation to a problem that is known to be difficult. The philosophy of the construction is to derive as much power as we can from the network-flow technique, which we know provides an optimal solution in the two variable case. The strategy is to group together occurrences in each mxcc into two subsets—black and white. The black occurrences as a whole act as a variable, and the white occurrences as a whole act as another variable. There are cohabitation arcs between black and white occurrences only at source nodes in the derived network-flow problem, just as in the two variable case, there are cohabitation arcs between the two variables only at a source node. Corresponding to the situation that two variables are in different mxcc's at a sink, we will construct the modification subgraph so that at a sink, a black and a white occurrence are never in the same mxcc.

In the absence of an implementation, it is possible to make only plausability arguments for this approach. The main argument we make is that there are not too many occurrences at a node of H_e and the modification subgraph never departs too far from the two variable case. Realize that if there are several occurrences at a node

of H_e they will have different variables, and there must be some assignment that caused the cohabitation. It is hard to imagine a real program where more than three of four variables have all been assigned together. Even with assignments generated by the compiler, say to model parameter passing, a half dozen occurrences at a single node of H_e seems an extreme number. There are several places in this and the next section where we invoke the smallness of mxcc-sets, usually to justify not worrying too hard about choices to be made. As we know from the two variable case and the proof of NP-hardness, as the number of occurrences at a node of H_e grows, so does the unlikelihood of finding a reasonable approximation.

We begin the construction of a modification subgraph M at a split. As in the two variable case, a split corresponds to an arc of M entering a sink. Here, there are arbitrarily many variables, and complications arise that were not seen previously. The first of these is that because a node of H_e may have several mxcc's that might be broken up by the insertion of move instructions, we must think of M as a subgraph of the mxcc-graph, not of H_e. (In two-variable split removal, the only part of the mxcc-graph where it made sense to put moves was where there were two active variables, and thus only one mxcc. This made the mxcc-graph correspond exactly to H_e.) Strictly speaking, M is not really a subgraph of the mxcc-graph, because there is only one arc of M entering a sink, where the mxcc-graph has a fan-out:

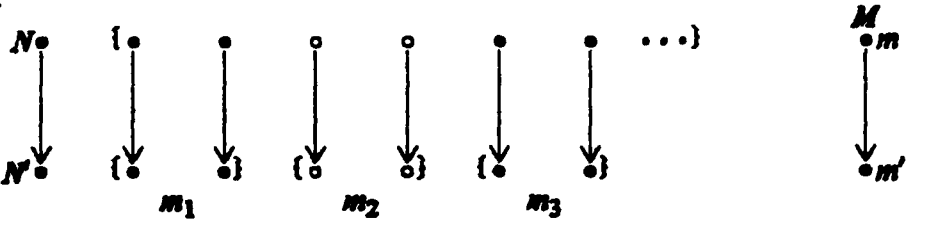


With this abuse of terminology understood, we continue to call M a modification subgraph, but of the mxcc-graph, not H_e.

A split is always associated with an arc A of H_e and a mxcc m belonging to the node at the beginning of A. The size of a split associated with A,m is one less than the number of mxcc-arcs leaving m and belonging to A (so the size is zero if there is no split). We describe a construction for M that reduces the size of the starting split, perhaps to zero. This same M may also, serendipitously, reduce the size of other splits. Inserting the move instructions corresponding to a cut of M may divide the

mxcc-graph into two connected components. Whether or not this happens, the total size of splits in the new mxcc-graph(s) will be less than the original. After several iterations of constructing a modification subgraph and inserting the move instructions corresponding to it, the total size of splits will be reduced to zero.

In the general case a split may take a single mxcc m at a node N of H_s to several mxcc's m_i at a subsequent node N. We shall choose at least one of the m_i to be black, and one to be white. By the smallness of mxcc-sets, there are probably only two mxcc's. In the unlikely event that there are more than two, black and white may be assigned to the others arbitrarily. The occurrences in each of the m_i receive the color of the m_i, and the occurrences in m are colored according to the occurrence they correspond to in one of the m_i. Initializing M thus produces a situation like this:



The M that we construct will correspond to the freedom we have in inserting move instructions that separate black from white along this split.

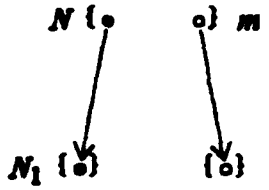
There may be occurrences in m that are last uses, and so will not be given colors by the above rule. We will eventually assign these occurrences either black or white, but on the basis of what can be seen between N and N', there is no reason to choose either one. It is convenient to assign last uses arbitrary distinct colors. As the construction of M proceeds, these colors may be merged together, or may be merged with black or white. But as we shall see, we never merge black with white.

Since colors correspond to cohabitation classes, the next step of the algorithm is to merge colors of occurrences that are connected by an intra-line cohabitation arc (for simplicity, assume that there is at most one CHB per node of H_s). This may result in some of the last uses becoming the same color, or black, or white. If this rule says to merge black with white, we don't. Rather, by analogy with the two variable case, m is labeled a source. In this case, M consists only of m, m', and an arc connecting them—there is no choice about where to put the move to reduce or remove this split.

Suppose that black and white do not intra-line cohabit at N. Then there remains the possibility of inserting a move instruction to separate black and white on a path leading to N. If this is done, then the black and white occurrences at N will end up in separate mxcc's after growing mxcc-sets, and of course this growth will propagate forward on all paths leaving N. Thus, before including in M anything before N, we investigate what would happen during the growth of mxcc-sets from N, if black and white occurrences were in separate mxcc's. This investigation is accomplished by a forward propagation of the colors of occurrences at N. (Colors other than black and white are not propagated, because occurrences with these colors are last uses.) As colors are propagated forward along a mxcc-arc, all of the non-first occurrences in the destination mxcc receive colors. The assignment of colors can be extended to all the occurrences in the mxcc by propagation along intra-line cohabitation arcs.

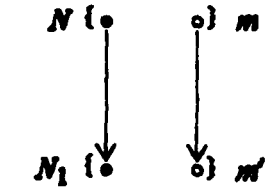
We first consider the case in which the occurrences of the mxcc are all black or all white. The entering mxcc-arc is not a place where inserting a move instruction will separate black from white, so this mxcc-arc is not included in M, and the scan does not continue from this point. However, for reasons that become clear later, the colors are left on the occurrences.

Let A be an H_a arc from N to N_1. Suppose that every mxcc-arc leaving m and belonging to A ends in an all black or all white node. This is called a *complete split*.



The forward step to N_1 taken here causes a situation that looks exactly like the original split. Naturally, we create a new sink m_1 for M, and connect m to it.

We next consider the case where the destination mxcc of the mxcc-arc receives both a black and a white occurrence.



We ask whether some pair of black and the white occurrences of m_1 are eventually-separate. If not, we will argue that, as an approximation, m should not be included in M. The part of the program reachable from N_1 does *not* reach a split for the mxcc m_1 (else the occurrences would be eventually-separate), meaning that black and white can cohabit as far as m_1 is concerned. It seems unlikely that an optimal split removal would separate occurrences that can cohabit. Thus, we terminate the forward scan, and do *not* include m_1 in M. Further, we argue that black should not be separated from white before N, for the reason that they would also be separated on paths starting from N, specifically, those leading to N_1 and beyond. Thus, we label m a source and terminate the construction of M. In this situation, as in the case where black and white intra-line cohabit at N, there is no choice about where to reduce the split.

The remaining case is that each black-white pair of occurrences at N_1 are in different mxcc's or are eventually-separate. In this case, we continue the forward scan from m_1. If a mxcc-arc leaving m_1 arrives at a mxcc that is not already colored, then we have the same cases that we had as we left m, and this is true in general as we scan forward. The new case is that we may encounter a mxcc that has already been colored.

The simplest and most pleasant case that arises in an already colored mxcc is that the cohabitation arcs along the mxcc-arc being scanned connect black with black, and white with white. In this case, the arc is included in M, and the forward scan continues along other paths. A related possibility is that the scan arrives back at m, and some of the black or white nodes propagate to last uses there. The colors of these last uses are merged with black or white, as required by the cohabitation arcs. As long as this can happen without an attempt to merge black and white, we have the simple pleasant case.

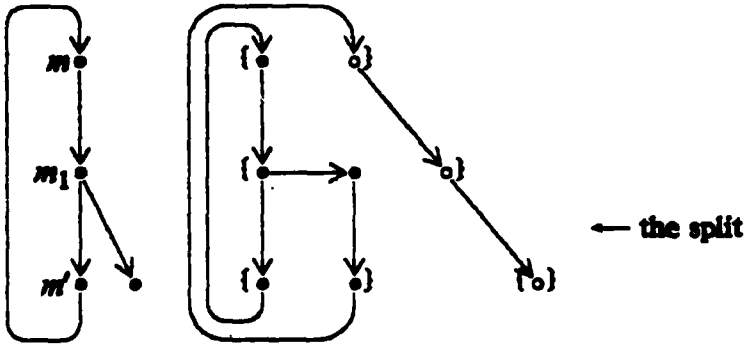
Suppose though, that a black occurrence is carried by a mxcc-path to a white occurrence at the same node. If this occurs, we have met a problem alluded to earlier—the propagation of colors (specifically, black and white) cannot be done consistently. This is called a *latent twist*, for a reason we now explain. Suppose we separate black from white (with a move instruction) on some path leading to the arc which caused the black-white merge. Then after split removal and growth of mxcc's,

we would have a twist, and there would have to be extra moves (exchanges) to resolve the inconsistency. Now, it is conceivable that all this might be part of an optimal inconsistency resolultion, but so unlikely that the construction of M excludes the possibility. The point is that we can dictate that the occurrences involved in a latent twist always cohabit, and still resolve the inconsistency—thus the term "latent". Following previous reasoning to ensure that occurrences in a latent twist cohabit, we label m a source, and delete from M all the structure that was added on the scan forward from m.

To summarize the forward scan from m, the effect is either to consistently assign colors to occurrences in every mxcc reachable on a forward path from m, stopping at all black and all white mxcc's, and to include all of this part of the mxcc-graph as part of M, or to label m a source, include none of this part of the mxcc-graph in M, and to terminate the construction of M (leaving it with only m, m', and the connecting arc).

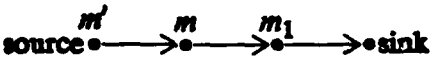
If the construction of M is not complete, the next is to scan backward from any mxcc m that has already been included in M. We take a backward step from a mxcc m, i.e., consider a mxcc-arc entering m, only when the coloring at m can be consistently propagated along all forward paths. After the forward scan from the top of the split (also called m), all of the nodes included in M during the scan, as well as the top of the split, enjoy this property.

Let m' be a mxcc having an exiting arc that enters m. If m' has been colored, the only possibility is that it is a mxcc that is part of a sink or is an all white or all black node at which the forward scan stopped. This bizarre case looks like the following (cohabitation graph on the right, corresponding mxcc-graph on the left):



Because mxcc-sets have been grown, back propagated occurrences arrive at a single

mxcc, so that if back propagation to a sink occurs, it will connect black and white. This is another version of a latent twist; we cannot propagate colors consistently. As we did before, we stop the propagation by labeling some node a source, in this case m'. This seems peculiar, because m' is also part of a sink. However, it must be correct, because it allows us the flexibility of inserting a move instruction between m' and m, m and m_1, and m_1 and the sink, each of which is a reasonable place to remove the split. The modification graph looks like this:



Like other latent twists, this probably is unlikely to happen in real programs.

The ordinary case is that the occurrences of m' have not already been colored. In this case, we include in M the mxcc m' and the mxcc-arc between it and m. The colors at m propagate backward, so we get colors at all occurrences of m' except the last uses (as at the top of a split) and occurrences that split off to some mxcc other than m (a new phenomenon in the general case). These occurrences are given new distinct colors. As before, we merge colors according to intra-line cohabitations at m', unless this would merge black and white, in which case we make m' a source, and do not continue a backward scan from it. Again as before we do not scan backward from m' until we scan forward. This forward scan is much the same as from the top of a split. If the scan encounters a latent twist or black and white occurrences that are not eventually-separate, then everything adjoined to M since the start of the scan from m' is excluded from M, and m' is labeled a source. A forward scan in the general case can encounter a source node (on the first forward scan, there were no sources). The reason that a node is labeled a source is that the split should be removed after that point, because otherwise a merge of black and white is implied. Thus, if a forward scan from m' encounters a source, we also conclude that m' should be a source, and as usual, we exclude from M everything that was adjoined since the start of the forward scan.

In addition to the possibility of encountering sources, there is another complication that we previously did not have to consider: some of the colors may be neither black nor white. We now review forward propagation, incorporating this extra generality. Suppose a mxcc does not receive a black-white pair of eventually-separate occurrences. If there are only black or only white occurrences, we have the

same situation as before, and take the same action. Otherwise, there is a dilemma. On the one hand, we cannot include the mxcc in M and continue the forward scan, because this part of the mxcc-graph would no longer represent places where an inserted move instruction would separate black from white. On the other hand if we merely quit scanning, there is the possibility that a merge of colors, made because of a construction elsewhere in M, might cause an eventually-separate black-white pair to appear in the mxcc, in which case we should have propagated forward.

The solution is to continue the forward scan merging colors according to cohabitation arcs, but *not* include any of the structure in M. We call this a *tentative* forward scan, and say that we tentatively include part of the mxcc-graph in M. Like the entire forward scan, it may be necessary to abort a tentative forward scan, for example, if a latent twist is discovered. If the beginning mxcc of a tentative scan has both a black and a white occurrence, then aborting the tentative scan causes an abort of the entire forward scan. Otherwise, a more benign approach to the abort may be taken. When we were considering the case with only black and white, the scan stopped when a mxcc was all black or all white. Thus, when aborting a tentative scan, we merge together all the colors at the beginning mxcc and eliminate the tentative part of M included in the scan. This may cause the mxcc to become all white or all black, or all some other color. The mxcc will still become a source if it is reached on a backward step, since its monochromicity would cause a merge of black and white.

We briefly review what can happen during a tentative forward scan. If a source or a latent twist is encountered, the tentative scan is aborted. If a step is taken to a mxcc that receives only one color, then even the tentative scan stops; the mxcc and the arc to it are not tentatively in M—no matter what merges of colors occur in the construction of M, a forward scan would not include these in M, and would not continue from here. Thus, we will maintain the rule that a mxcc tentatively in M always has distinct colors, just as mxcc's in M always have black and white. However, merges of colors later in a tentative scan, or even later in the construction of M, can cause this to be violated. Merges of colors must therefore be accompanied by a check on whether they violate this rule. If so, the mxcc's tentatively in M and arcs to them are no longer tentatively in M. It is as if the tentative scan had never gone beyond this point. An efficient implementation of the check on merges and possible

undoing of the tentative scan is a programming problem not considered here. If a mxcc becomes all black or all white, it is possible that a mxcc leading to it now has a complete split. This possibility must be checked, and a sink added to M if it occurs.

It is possible for a forward step in a tentative scan to arrive at a mxcc that is already in M. We merge colors along the mxcc-arc as usual; this may detect a latent twist, aborting the tentative scan. If not, the merges guarantee an eventually-separate pair of black-white occurrences at the mxcc from which the forward step was taken, and in fact at any node on a path from the beginning of the tentative scan. Thus, if the tentative scan terminates without aborting, we check to see if there is now an eventually-separate black-white pair. If so, everything in the tentative scan that has such pairs is included in M. The test of M will consist of pieces, each having a root mxcc with the property that all the colors of a piece appear in the root mxcc. (If none of the tentative scan is included in M, the root mxcc is the beginning mxcc of the forward scan.) Because of the tentative scan, any merge of colors of a root mxcc will not lead to a latent twist in the tentative part of M forward of the root. If all the colors are merged to black or all to white, that tentative part of M will return to its unscanned state. If some become white and some black, then some of the tentative part will be included in M, up to a complete split, or to mxcc's that become roots of smaller tentative parts of M.

Let us suppose that we finish a tentative scan from a mxcc, but that the mxcc remains tentatively in M. Then we leave everything tentatively in M. This means that another forward scan may find a mxcc tentatively in M. If the merges along the arc detect a latent twist the forward scan is aborted (if it is a tentative scan, the tentative part is aborted). Otherwise, the merges are completed. If the forward scan was in a non-tentative part, it may continue. If the forward scan was tentative, it need not go beyond this point. Because the tentative part of M is closed in the forward direction, a backward step will never reach a mxcc tentatively in M.

After M is closed under backward and (perhaps tentative) forward steps, we have essentially completed its construction. It remains to decide what to do with the tenative parts of M, but this is more naturally considered in the next section. Omitting the complicated details of forward scan, we now summarize this section by sketching

Algorithm 8.4 Construct a modification subgraph *M*

```
Initialize M at a split
Scan forward from the top node of the split
for A ← some mxcc-arc not in M entering a non-source mxcc in M
  m ← the end of A not yet in M
  Include A,m in M
  Scan forward from m
```

The occurrences in each mxcc in *M* or tentatively so are colored. Call a non-sink, non-source mxcc an *interior* mxcc. Each interior mxcc has occurrences with distinct colors. An interior non-tentative mxcc has at least one black and one white occurrence. Two occurrences belonging to interior mxcc's and connected by a cohabitation arc (inter-line or not) have the same color.

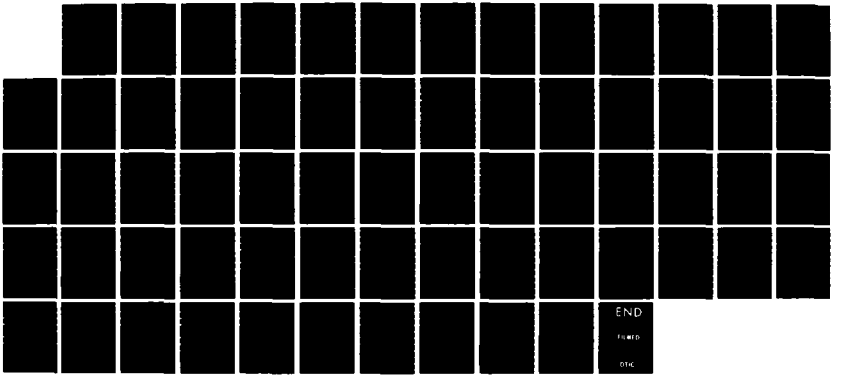
A final issue to be discussed is the assignment of costs to the arcs of *M*. Each arc *A* of *M* corresponds to some set of cohabitation arcs. For arcs in a strongly connected component of *M*, we use a cost of infinity, for the reasons outlined in section 8.1. For other arcs, we use the minimum of the costs on the associated cohabitation arcs, following our usual philosophy of optimism in choosing costs. Since all the cohabitation arcs correspond to the same arc in the flowgraph, it is likely that the costs will all be the same. This breaks down only when the variables in a cohabitation class are asymmetric in some respect. For example, in inter-region compilation, some of the variables may be assumed to be in registers, and others not. The only reason that two such variables are considered to be cohabiting is the technique we have been using to resolve an inconsistency.

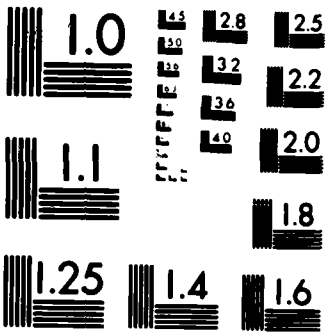
Our construction of *M* systematically excludes one type of cohabitation arc: an intra-line cohabitation arc at any source node. During the construction of a kernel region, such arcs may be given very low costs because we have some trick in mind for breaking the arc; thus we really must include this information in *M*. To do so, we can add a new source to *M* and a new arc leading from it the the old source, where the cost on the new arc is that of the intra-line cohabitation. The old source is then relabeled as a non-source. In this way, *M* can be made to reflect the information about inexpensive intra-line cohabitations. (To simplify exposition, this matter was not mentioned in section 8.1).

As an example of the techniques of this section, we consider the construction of a

AD-A151 549 THE DEVELOPMENT OF A PROGRAMMING SUPPORT SYSTEM FOR RAPID PROTOTYPING TASKS 2 AND 3(U) SOFTWARE OPTIONS INC CAMBRIDGE MA JAN 85 SO-01-85 N00014-82-C-0173 2/2

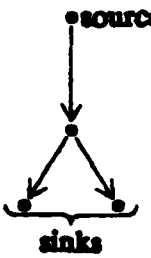
UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

modification subgraph for the split-removal problem generated by the NP-hardness proof of 8.2. Choosing a split in the mxcc-graph corresponds to choosing an arc in the graph that was to be colored. Recall that the nodes of this graph correspond to variables in the split-removal problem. The construction for M will assign the same color to every occurrence of the same variable, and different colors to occurrences of different variables. There will be one other sink besides the initial one. There is no tentative part of M, because each of the ones leaving the loop corresponds to a (non-exact) split where each mxcc-arc has only one occurrence. After shrinking arcs with infinite cost (those in the loop), M will have the form:



8.5 The General Case of Split-Removal

In the previous section, we saw how to construct a modification subgraph, and obtain an assignment of colors to its occurrences that is preserved along cohabitation arcs. If the resulting construction has only the colors black and white, then we have a situation like that of 8.1, even though we did not begin with two variables. We use the techniques of that section to insert move instructions that remove the splits between black and white. Inserting these involves modifying the cohabitation relations, and thus the mxcc-sets. After regrowing mxcc-sets, there may still be remaining splits, unlike the two variable case. If so, a new modification subgraph is constructed, and the process is repeated. By the remarks made at the beginning of the previous section, the number of times this process can be repeated is limited by the total size of all splits in the original mxcc-graph, and is expected to be small.

The real purpose of this section is to discuss the case in which there are colors other than black and white. We first discuss the case where M has no tentative parts. We will still insert move instructions on a minimum cut of the modification subgraph; all mxcc's below this point will be divided into two mxcc's—one mxcc having no black occurrences, and one having no white occurrences. Occurrences with other

colors will be in one mxcc or another. This must of course be done so that the cohabitation arcs are consistent with the mxcc-arcs, but there is still some choice for how to do this. Suppose that after insertion of move instructions, a red occurrence is in the same node as a black occurrence at some mxcc. By following cohabitation arcs forward and backward, we will see that red occurrences must always be in the same mxcc as a black occurrence. Thus, we may view red as being merged into black. Conversely, we may choose to merge any non-white occurrence into black, or vice versa, and obtain a valid mxcc-graph after insertion of move instructions.

The choice of merging a color with black or with white will clearly lead to different mxcc-graphs after insertion of move instruction. One choice is likely to lead to a more optimal overall solution to split-removal than another. How can we figure out what to do? We can offer only a heuristic and a plausibility argument. Before doing so, we observe that the construction of the modification subgraph produces as many distinct colors as possible, within the constraint that colors follow cohabitation arcs. Once a minimum cut is chosen, we are worried only about cohabitations below the cut, and are not constrained by any merges of colors that occurred because of cohabitations above the cut. To give the heuristic maximum flexibility, the effect of these merges is undone, perhaps yielding even more distinct colors.

We are guided by the idea of coloring the eventually-separate relation. Assume that at some mxcc, each of the occurrences has a distinct color. Momentarily forgetting about these colors, use different colors to (minimally) color the eventually-separate relation on the mxcc (ignoring self-loops). If the chromatic number is less than the number of nodes, then two nodes have the same color. Remembering again the original set of colors, we observe that the black and the white occurrence are eventually-separate (by construction of M), and so will receive different (new) colors. The basic heuristic is this:

> If two occurrences receive the same *new* color, then merge the corresponding *original* colors.

This will never merge black with white. The plausibility argument for the heuristic involves the consideration of what happens after growing mxcc-sets in the new mxcc-graph. If the eventually-separate relation is divided into two pieces in this way, then the sum of the chromatic numbers of the pieces will be equal to the chromatic

number of the original. In other words, the heuristic insures that the number of cohabitation classes does not increase, as it might for some other division of the eventually-separate relation.

It may of course happen that several of the occurrences in a mxcc already have the same color, either because of a cohabitation in *M*, or because they were merged together by the heuristic. We do not renege on making these colors identical, rather, we ask whether any further merging is allowed by the eventually-separate relation. Put differently, form an induced relation on the original colors at a mxcc, defining two colors to be eventually-separate if any two occurrences with those respective colors are eventually-separate. We may color the induced relation with new colors, and still apply the same heuristic.

It is necessary to apply the heuristic at most once on each node. After having done so, the induced relation will be a complete graph and will remain so after subsequent merges of colors. A complete graph has a chromatic number equal to the number of nodes; applying the heuristic step will not merge any colors. Conversely, an induced relation that is not a complete graph has a chromatic number less than the number of nodes, so applying the heuristic will cause a merge of colors.

After the heuristic is applied at each node, we know that the induced relation is a complete graph everywhere, but there is still the possibility that there are non-white, non-black colors. How should these be merged with black and white? It is difficult to formulate a further rule on the basis of what is seen in *M*, partly because a complete graph is symmetric on the node set. Experience may suggest further heuristics, but smallness of mxcc-sets indicates that the proposed heuristics result in only two colors in almost every practical case. A first implementation can choose the remaining merges arbitrarily.

We now turn to the problem of deciding what to do with the tentative parts of *M*. We use exactly the same idea of finding a minimal coloring and using it to induce merges. Given a root mxcc, it would seem reasonable to choose a mxcc leading immediately to a root mxcc as a place to start. As before, this is only a heuristic, and is not guaranteed to eliminate all the tentative parts of *M*. Remaining choices can be made arbitrarily.

It is interesting to apply this heuristic to the split-removal problem generated by NP-hardness proof of section 8.2. The eventually-separate relation for nodes in the loop will be the graph that was to be colored. The modification sub-graph will assign the distinct colors to each node, among them black and white; the heuristic will merge the colors of occurrences that are colored the same. *Regardless* of how we group the merged colors with black and white, we get two graphs, the sum of whose chromatic numbers equals the original. The split-removals that remain after the first split-removal will continue to obey an optimal coloring of the split-removal problem. Thus, the heuristic optimally solves the original split-removal problem, assuming that we can minimally color a graph.

9. Untwisting

9.1 Two Variable Untwisting

We saw in the previous chapter how to remove splits. If there is any remaining inconsistency, we know by Theorem 7.5 that twists account for it. The simplest example is the conditional exchange that we discussed earlier. In this section we will consider the problem of optimally untwisting a mxcc-graph with only two variables involved. Let us use H_*^n to denote the subgraph of H_* induced by the subset of nodes that have exactly n mxcc's (n = 1 or 2 in this section). There will be arcs from H_*^2 to H_*^1 (these are in neither subgraph), but because there are no splits, there will be no arcs from H_*^1 to H_*^2. By definition, a twist has an associated simple undirected cycle. We first concern ourselves with the case in which the cycle is in H_*^2. As we shall see, these are the inconsistencies that are resolved by exchanges. Our approach is to first find *any* set of exchanges that will work; these will be placed on arcs of H_* so we need merely identify the subset of arcs. This is done by fixing the ordering of the pair of mxcc-sets on some node of H_* propagating this ordering, and marking arcs where trouble occurs. A bit more formally, one can use the following depth-first-search algorithm.

Algorithm 9.1 Order mxcc-sets M_1, M_2 on node N

```
Mark node N as "seen"
Order M1 before M2 on N
for A ← each H*2 arc leaving N
    N' ← the other end of N
    Mi' ← node from traversing mxcc-arc from Mi along A, i = 1 and 2
    if N' is seen
      if M1' is after M2' on N' then mark A
    else
      Order mxcc-sets M1', M2' on node N'
```

This subroutine is used in:

Algorithm 9.2 Mark exchange arcs

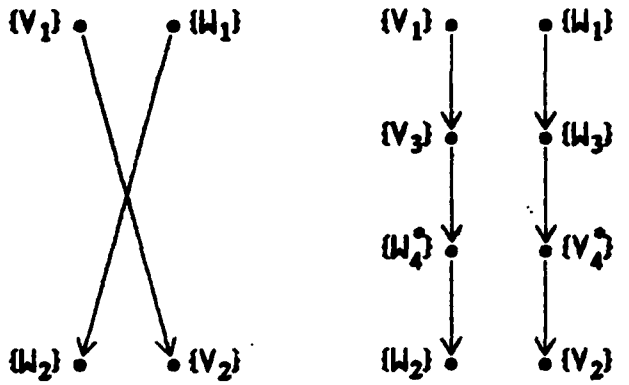
```
Initialize nodes of H*2 as not seen, arcs as not marked
for N ← each node of H*2
    if N is not seen
      M1, M2 ← the two mxcc-sets for N
      Order mxcc-sets M1, M2 on N
```

As we hinted earlier, the point of this is:

Theorem 9.1 If exchange instructions are placed on the arcs marked by the above algorithm, the cycles for any remaining twists go through nodes in H_*^1.

Proof We observe first that if the above algorithm marks no arcs, there is no twist in H_*^2. In this situation, a mxcc-path never goes from the first mxcc on one node to a second mxcc on the next; thus it can never return to a different mxcc on the same node.

Next, we examine the effect of placing an exchange on a marked arc of the mxcc-graph. An exchange instruction requires four occurrences in order to be properly represented by the cohabitation relation: two last uses (one for each variable) and two generations (one for each variable). Each last use is required to cohabit with the generation of the other name—this is the semantics of exchange. In the before and after pictures below, the exchange instruction involves occurrences with subscripts 3 and 4:



If the algorithm is run with the modified mxcc-graph, no arcs of H_*^2 will be marked, and by the earlier remark, no twist will lie entirely in H_*^2. □

We consider twists that do not lie entirely in H_*^2. It is impossible for a twist to lie entirely in H_*^1 because there are not distinct mxcc's at any node, by definition of H_*^1. Any twist thus crosses the boundary from H_*^2 to H_*^1 with the H_*-arc in that direction. Since the algorithm for growing a mxcc-set has been applied, the only way that the number of mxcc's can decrease is if one of them dies (i.e. all the occurrences in the mxcc die). Hence, there is only one mxcc-arc along this arc of H_* and either it leaves from the first mxcc on the H_*^2 node, in which case we call it a type-1 arc, or from the second mxcc, where it is a type-2 arc. This partitions the boundary arcs between H_*^2 and H_*^1.

The idea for identifying the remaining twists is to propagate the type from a boundary arc along forward paths, (these necessarily remain in H_*^1), marking arcs where this cannot be done consistently. First, we present the depth-first-search part, then the initiator.

Algorithm 9.3 Propagate type *i* along *A*

```
N ← node pointed to by A
if N has a type
    if the type of N is not i then Mark A
else
    Make N by type i
    for A' ← each arc leaving N
      Propagate type i along A
```

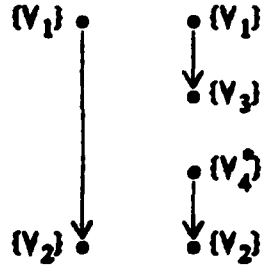
Algorithm 9.4 Mark move arcs

```
Initialize nodes of H*1 to have no type
for N ← each node of H*2
    for A ← each boundary arc leaving N
      i ← the type of A
      Propagate type i along A
```

The name of the latter algorithm anticipates the following result.

Theorem 9.2 If move instructions are inserted on the arcs marked by the above algorithm, there are no twists through nodes of H_*^1.

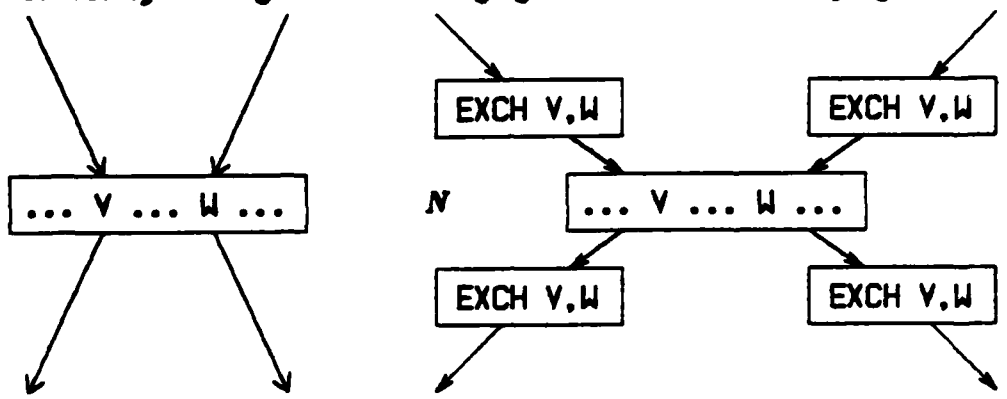
Proof As with the previous result, we first note that if the algorithm marks no arcs, there is no twist; any mxcc-path from H_*^2 into H_*^1 and back stays on the same type of mxcc (first or second). Then, we look at the effect of a move instruction on the mxcc-graph.



In other words, the arc in H_* is effectively removed, so there is not the possibility of marking it.

The algorithms and theorems of this section have shown how to remove twists by the insertion of exchanges and moves. So far, we have paid no attention to how to do this optimally. In order to pursue this question, we will investigate the family of

all possible insertions of exchanges and moves that will resolve the inconsistency. We begin by looking at a local change in the insertion of exchanges and moves. Their are two basic observations. The first is that a node of H_s can be completely surrounded by exchanges without changing the correctness of the program:



The code of node N will find V and W stored in opposite places in the two cases, but as long as this is fixed it is clear that the left side is correct if and only if the right side is. Although this picture is drawn with nodes and arcs of H_s^2 in mind, it is actually correct if some of the arcs are boundary arcs, or if N is in H_s^1 and the arcs are boundary arcs on H_s^1 arcs. We make the convention that the exchange swaps the contents of the two memory locations corresponding to the two mxcc's and optimize the exchange to a move when it appears on a non-H_s^2-arc.

The second basic observation is even simpler to understand: two consecutive exchanges reduce to nothing. Two moves along a non-H_s^2-arc also cancel. No pictures are necessary to illustrate this.

We can combine the two observations into a single operation. Recall that the algorithms marked certain arcs, upon which either exchanges or moves are inserted, depending on membership in H_s^2. Pick a node N. Suppose we increase the number of marks on each node by one (following the first observation), and take all marks off of a doubly marked arc (following the second observation). This amounts to complementing the marks on the arcs incident to N, and leaves us with a marking that will resolve the inconsistency (once exchanges and moves are inserted). Further, this operation completely disregards direction in H_s, leading us to the following nomenclature (analogous to that for Petri-nets).

Definition An *undirected marked graph* is an undirected graph, together with a

function from the arcs to {0,1} (the *marking*). To *fire* a node is to complement the value of the function on arcs touching it. □

Theorem 9.3 Given a split-free ms-partition H_s an underlying mxcc-graph, and a marking of the arcs by the above algorithms of this section. Any other marking of the arcs will remove the splits if and only if it can be obtained from the first by firing some sequence of nodes.

Proof We have seen above that any sequence of firings leads to a marking which will remove the twists. Conversely, for any marking and undirected cycle in H_s let the *parity of the cycle* be the parity of the number of marks along arcs in the cycle. We claim that for any fixed cycle, the parity of markings that remove a twist is the same–it is one if the cycle gives rise to a twist, and zero otherwise. Thus, the parity of the sum of two such markings is zero. Since this holds for every cycle, we can two-color the nodes of H_s so that nodes connected by an arc with no mark from either marking, or marks from each are the same color, while nodes connected by arcs with precisely one kind of arc are different colors.

Now, start with one of the markings, and fire all the nodes of a given color. The order is irrelevant: arcs with no marks or both marks stay the same, while arcs with precisely one kind of mark receive the other kind of mark. Hence, any marking that removes twists can be reached by firing nodes, starting with any other such marking. □

Given a frequency on an arc, we can calculate how much it would cost to place an exchange there (see section 2.4). This defines the *cost* of an arc of H_s. Starting with this cost, we have the further

Definition Given an undirected marked graph with costs on its edges, the *cost of a marking* is the sum of the costs of marked edges. Given a marking, a *minimal equivalent marking* is one that can be reached from the original marking by firing nodes, and has cost no greater than any other such marking.

□

The next section will consider the problem of finding a minimal equivalent marking. We summarize this section by

Algorithm 9.5 Optimally remove twists

Mark exchange arcs of H_e
Mark move arcs of H_e
Viewing H_e as a marked undirected graph find a minimal equivalent marking
Insert exchanges and moves according to the minimal marking

9.2 Undirected Marked Graphs

In the previous section, we reduced the problem of optimal untwisting in the two variable case to an optimization problem on undirected marked graphs. In this section, we show that this problem is NP-hard in general, but that the problems arising in practice are likely to be easily solved. We begin with a simple observation used in both the negative and positive results.

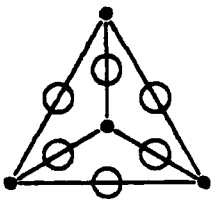
Lemma 9.1 Let G be a undirected marked graph, with a set of nodes $\mathfrak{N}$. Let F be any subset of nodes. Then firing F and firing $\mathfrak{N} - F$ lead to the same marking.

Proof The marking on an arc changes only when one of its ends is in the firing set and one is not. This property is invariant when F is replaced by $\mathfrak{N} - F$.

□

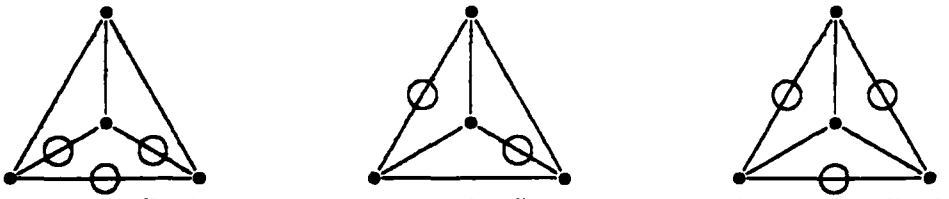
The proof of the NP-hardness result is by reduction from 3-colorability ([3]). The lemma below considers the gadget that will be used for each of the nodes of the graph to be 3-colored.

Lemma 9.2 Consider the "tetrahedral" undirected marked graph:



Assume that the center node does not fire. Then a minimal equivalent marking can be obtained by firing any pair of other nodes, and only by firing a pair.

Proof Since the graph is symmetric on the outer three nodes when we consider the inner node distinguished (unfired), we just investigate what happens when firing 0, 1, 2, or 3 nodes. The picture for 0 fired nodes is unchanged from the above. The other three pictures are:

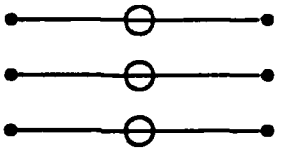


The middle picture has a cost of two, the others either three or six.

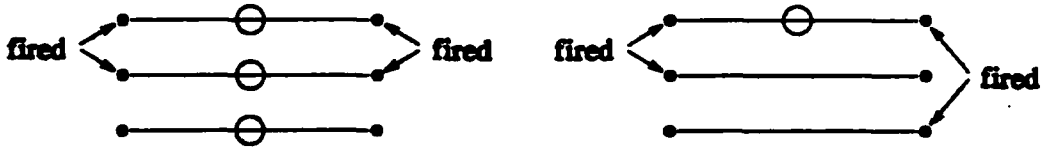
□

Next, we consider the gadget that will be used for each of the arcs in the graph to be 3-colored.

Lemma 9.3 In the undirected marked graph below, assume that a pair of nodes on the left fire, and a pair of nodes on the right fire. Subject to this condition, a minimal equivalent marking is obtained only if the fired pairs do not correspond horizontally.



Proof By symmetry, there are essentially only two cases, when the fired pairs correspond, and when they do not.



□

Using these two types of gadgets, we obtain

Theorem 9.4 The problem of finding a minimal equivalent marking is NP-hard, even with uniform weights on the arcs.

Proof Given a graph to be 3-colored, form a node gadget for each of its nodes, as in Lemma 9.2, all sharing a common central node. Label the other three nodes with red, blue, and yellow. Where nodes are adjacent in the graph to be 3-colored, connect the red-red, blue-blue, and yellow-yellow pairs of the corresponding node gadgets with the arc gadget of Lemma 9.3.

By Lemma 9.1, we may assume that the common central node of the node gadgets

does not fire. We wish to arrange weights so that in any minimal equivalent marking, a pair of nodes in each node gadget will fire. Thus, we make the cost of arcs of node gadgets high compared to the cost arcs of the arc gadgets (red-red, ...). Specifically, if there are α arcs in the graph to be 3-colored, let the cost of a node gadget arc be $3\alpha+1$, and the cost of the other arcs be 1. Suppose that there are ν nodes in the graph to be 3-colored. By Lemma 9.2, the cost of the marks in node gadget arcs alone will be $2\nu\cdot(3\alpha+1)$, and this will be achieved only when a pair from each node gadget is fired. Assume that a firing set does not fire a pair from each node gadget. By changing the firing on that node gadget alone, we can reduce the cost by $3\alpha+1$, and at worst, increase the cost on arc gadget arcs by 3α, a net reduction of at least one. In short, the cost of a minimal equivalent marking, restricted to node gadget arcs, is fixed at $2\nu\cdot(3\alpha+1)$, and we worry about the "excess cost", which is between α and 3α.

Suppose we find a minimal equivalent marking. This chooses a pair of colors from each node gadget. Mix these colors together, and assign this color to the graph to be 3-colored, i.e., red and blue yield purple, etc. If the excess cost is exactly α, then by Lemma 9.3, the graph to be 3-colored has been successfully 3-colored. On the other hand, if the original graph can be 3-colored, say with orange, green, and purple, the colors may be put through a prism to obtain fired pairs on the node gadgets, with a total excess cost of exactly α. Thus, the original graph is 3-colorable if and only if the minimal equivalent marking in the derived graph has cost $2\nu\cdot(3\alpha+1)+\alpha$. This proves that finding a minimal equivalent marking is NP-hard.

It remains to be shown that we can make the construction using uniform weights. The point is that we can replace each node gadget arc with $3\alpha+1$ (marked) arcs, and obtain precisely the same behavior and still have an undirected marked graph whose size is polynomial in the size of the graph to be 3-colored.

□

Observe that in the construction used to prove NP-hardness, every single arc is marked. Recalling the construction of the undirected marked graphs used in untwisting (section 9.1), it seems likely that comparatively few arcs will be marked. The rest of this section concerns a technique that works well with few marked arcs. We begin with a result that allows us to decompose the problem to a certain extent.

Theorem 9.5 Let G be an undirected marked graph. The cost of a minimal equivalent marking is equal to the sum of the costs of minimal equivalent markings of the biconnected components.

Proof We use induction on the number of articulation points. If there are no articulation points, either G is biconnected, in which case the result is a tautology, or else G consists of a single arc touching two nodes. If this arc is not marked, it is its own minimal equivalent marking; if it is marked, fire one of the nodes to get an equivalent marking with cost zero, and therefore minimal. This proves the result when the number of articulation points is zero.

Assume G has an articulation point n. Then G is the union of some number of subgraphs $G_1,...G_k$, whose pairwise intersection consists only of the node n. Each G_i will have fewer articulation points than G, so we may apply the result inductively to each of these. Let F_i be the firing sets for each of the G_i that achieve a minimal equivalent marking. By the Lemma 9.1, we may assume that F_i does not contain the node n. This proves that the sum of the costs of minimal equivalent markings is greater than or equal to the cost of a minimal equivalent marking of G. On the other hand, let F be a firing set that achieves minimal cost for G. Restricted to G_i, F leads to an equivalent marking for G_i, so the cost of a minimal equivalent marking for G is greater than an equal to the sume of the costs of minimal equivalent markings. Equality is established.

□

Since biconnected components can be computed in linear time (see [5]), this is a useful reduction. In fact, this can be done at the same time that the arcs are marked (see 9.1), because both are depth-first-search algorithms.

As we shall see, the techniques of flows in networks are useful in looking for a minimal equivalent marking. The classic reference is [1]; we introduce some standard terminology.

Definition Let F be a subset of the nodes of G. The *boundary* of F, denoted ∂F, is the set of arcs with one end in F and the other in $\mathfrak{N} - F$. A *cut* is a set of arcs that is the boundary of some set of nodes.

□

In network flow theory, numbers assigned to arcs are called capacities, because of the

original application of the theory. In the present context, they are costs, and we will continue to call them that. We shall also extend the meaning of the term:

Definition The *cost* of a set of arcs (typically a cut), is the sum of the costs of its elements. Given a marked graph G, the *cost* of a set of nodes F is the cost of the set of marked arcs after F is fired.

□

The following result is the connection to network flows.

Theorem 9.6 Let G be an undirected marked graph and H the subgraph subtended by unmarked arcs. Let F be a firing set achieving a minimal marking. For every marked arc in G, label an end as either a source or a sink, according to whether the endpoint is *or is not* in F. Let C be a minimum cut of H between sources and sinks, and let S be the set of arcs incident upon two sources or two sinks. Then

$$\text{cost}(F) = \text{cost}(S) + \text{cost}(C)$$

Proof Let C be a cut set, and let F' be a set of nodes such that $\partial F' = C$ (without loss of generality F' will contain all of the sources and none of the sinks). Fire all the nodes of F'. This will remove all marks on arcs incident upon a source and a sink, and will put marks only on the arcs in C. Thus

$$\text{cost}(F) \leq \text{cost}(F') = \text{cost}(S) + \text{cost}(C)$$

The inequality is by the minimality of cost(F).

Conversely, given F, let $C' = \partial F$, so that any path in H from a source to a sink must include an arc of C'. Firing F adds marks only to arcs of C'. Thus

$$\text{cost}(S) + \text{cost}(C) \leq \text{cost}(S) + \text{cost}(C') = \text{cost}(F)$$

The inequality is by minimality of cost(C), and the result follows.

□

A minimal equivalent marking can trivially be found in time exponential in the number of nodes of G. Because of network flow theory, minimum cuts can be found in polynomial time, so the above result means that we can find a minimal equivalent marking in time exponential in the number of marked arcs of G. In practice, this is probably good enough, since this number is most likely one or two. A number of heuristics can be devised, based on consecutive applications of max-flow-min-cut. One of the simplest is the following, which uses a greedy approach to orienting marked arcs.

Algorithm 9.6 Approximate minimal equivalent marking

All marked arcs are initially unoriented.
for $A \leftarrow$ marked arcs in order of decreasing cost
  choose an orientation for A to minimize min-cut
  if cost of min-cut < cost of oriented arcs
    use cut to achieve lower cost marking
    restart the algorithm

9.3 Permutation Labeled Graphs

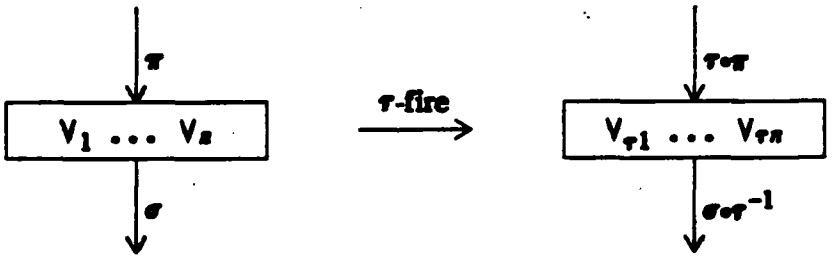
We have considered the problem of untwisting when there are only two variables. The techniques actually apply to the more general case in which each node of H_∞ has at most two mxcc's. This and the next section consider the case in which a node of H_∞ has any number of mxcc's. (We always assume that splits have been removed from H_∞.) We know that the two variable case is hard (Theorem 9.4), so the general case must rely on heuristics as well. However, the added difficulties of this case do not present any real discouragement, not because they are easy to handle, but because of what must be their great rarity. The conditional exchange is a reasonably natural example of how a twist can arise from a real program. I know of no similarly natural example involving three or more variables. The best example I can devise is a sort program that works on three scalar variables, written so that a value is not moved until its precise point in the ordering is known. Such a program would present three-variable twisting; why anyone would write such a program is not clear.

Nevertheless, for completeness, we consider general untwisting. Recall the notation H_∞^n, denoting the subgraph of H_∞ induced by nodes having exactly n mxcc's. In this section, we will assume that $H = H_\infty^n$, i.e., every node of H_∞ has exactly n mxcc's. This restriction is removed in the next section. The first task is to order mxcc-sets on nodes of H_∞ and to make this ordering obey cohabitations along arcs of H_∞ as far as possible. When this became impossible in the two-variable case, we marked the arc. Here, the mark must carry more information; to specify it exactly is to give the permutation on n elements which corresponds to the map induced by the cohabitation arcs along the arc of H_∞. From this point of view, the unmarked arcs of the two variable case are labeled with the identity permutation; the marked arcs are given the only other permutation on two elements, namely, a transposition. To make this explicit in several variables, consider an algorithm like 9.1, and let us look at

what might happen along an arc to an already seen node, i.e., one at which the ordering of mxcc sets is fixed. For $n = 3$, there are essentially two cases:

Representing the permutations as products of cycles, and numbering the mxcc-sets from left to right, we have the permutations (2 3) and (1 3 2)—here each product has only one non-trivial cycle. In the product of cycles representation, the identity permutation can be represented by a null product, in keeping with the unmarked nature of arcs with this permutation. We will exploit the fact that if the cycles are disjoint, then the representation is unique, up to ordering in the product and cyclic permutations in each cycle ([2], Theorem 5.1.1).

In the two variable case, we observed that a node could be completely surrounded by exchanges, and the code of the node adjusted to preserve the correctness of the program. In the present context, this observation must be made in terms of permutations. Rather than merely "fire" a node, we are able to τ-fire a node, where τ is a permutation (indices denote variables, not occurrences):



In words, suppose we follow the permutation π with the permutation τ. On entering the node, the variable that used to be in position i will now be found in position $\tau(i)$; changing the names in the node will cause the program to behave in the same way as before. To insure correctness after leaving the node, the effect of τ must be undone, i.e., we must apply τ^{-1}, followed by σ. The effect of going through both arcs is $(\sigma \circ \tau^{-1}) \circ (\tau \circ \pi) = \sigma \circ \pi$, just as before. In the case of multiple input arcs, we must treat each one as above; similarly for output arcs. The notion of τ-firing leads to the basic definition of this section.

Definition A *permutation-labeled-graph* is

- a directed graph G, and permutation group S_n

- a labeling for the arcs of G drawn from S_n and
- a function that gives the cost of labeling a given arc with a given permutation

The *minimal equivalent labeling problem* is to choose for each node N of G a permutation τ_N such that the set of firings $\{\tau_N\}$ minimizes the sum of the costs of the resulting labels on arcs of G.

□

In order to make any progress at all on this problem, we must make some restrictions on the nature of the cost function. These are based on the following group-theoretic notion.

Definition Given $\sigma \in S_n$, the *signature* of σ is obtained by expressing σ as the product of disjoint cycles and forming the multiset of the lengths of the cycles.

□

Thus, the signature of (1 3 2) is {3}, while that of (1 4)(2 3) is {2,2}. We shall make a *reasonable cost assumption* that the cost of a permutation on any given arc depends only on its signature, that smaller signatures (in the sense of containment) have lower cost, and that shorter cycles have lower cost. We are then able to exploit group-theoretic result.

Lemma 9.4 The signature of an element of S_n is invariant under conjugation and inversion. In particular, signature($\sigma \circ \tau$) = signature($\tau \circ \sigma$) for any $\sigma, \tau \in S_n$.

Proof For conjugation, see [2], Theorem 5.4.1; the result for inversion is clear by inspection. For commutativity, $\sigma^{-1} \circ (\sigma \circ \tau) \circ \sigma = \tau \circ \sigma$.

□

Suppose we are given a graph whose arcs are labeled by permutations represented as products of disjoint cycles, thereby giving the graph a certain cost. We ask first whether the cost can be reduced by picking a transposition τ and either τ-firing a node, or leaving it unfired. We show that this problem reduces exactly to the undirected marked graph problem considered previously, regardless of how complicated permutations on the arcs are.

The undirected mark graph that we construct has exactly the same graph structure as the one with which we started, except that the arcs are considered undirected. The problem is choosing the costs. Consider an arc with permutation σ. If we τ-fire

one end of the arc, we get the permutation $\tau \cdot \sigma$, while if we τ-fire the other end of the arc, we get $\sigma \cdot \tau^{-1}$. Since τ is a transposition $\tau^{-1} = \tau$, and by Lemma 9.4, $\text{cost}(\tau \cdot \sigma) = \text{cost}(\sigma \cdot \tau)$. Thus, if we fire one end of the arc, we get the same difference in cost, no matter which end we fire. Further, if we fire both ends of the arc, we get the permutation $\tau \cdot \sigma \cdot \tau^{-1}$, whose cost is unchanged from that of σ, again by Lemma 9.4. Thus, in the undirected marked graph, the weight of this arc is $|\text{cost}(\sigma) - \text{cost}(\sigma \cdot \tau)|$. If cost (σ) > cost $(\sigma \cdot \tau)$, we mark the arc, so firing one end causes a reduction in cost; otherwise, we leave the arc unmarked, so a single firing causes an increase.

It is natural to want to limit the number of τ's that can be considered. Improvement can occur only if $\text{cost}(\sigma \cdot \tau) < \text{cost}(\sigma)$ for at least one σ. This will hold by the reasonable cost assumption if τ is among the disjoint cycles of σ. More generally, it holds if the elements of τ are adjacent in some cycle of σ. This is because shorter cycles have lower cost, and:

$$(1\ 2)(1\ 2\ a \ldots b) = (2\ a \ldots b)$$

If σ does not involve the elements of τ, then clearly $\text{cost}(\sigma \cdot \tau) > \text{cost}(\sigma)$, because the former signature is larger. The equivocal cases arise because of the following identity (assume $a_1, \ldots, b_1$ are disjoint from $a_2, \ldots, b_2$):

$$(1\ 2)(1\ a_1 \ldots b_1)(2\ a_2 \ldots b_2) = (1\ a_1 \ldots b_1\ 2\ a_2 \ldots b_2)$$

By considering the move sequences necessary to implement the permutations, we would expect that

$$\text{cost}((1\ a_1 \ldots b_1\ 2\ a_2 \ldots b_2)) < \text{cost}((1\ a_1 \ldots b_1)(2\ a_2 \ldots b_2)), \text{ usually.}$$

The only place this might fail is if the machine has an exchange instruction, in which case we might have, for example,

$$\text{cost}((1\ 3)(2\ 4)) < \text{cost}((1\ 3\ 2\ 4))$$

In any case, once the structure of the machine is known, it is possible to eliminate a large number of potential τ's at the outset. We also observe that several τ's may give rise to the same undirected marked graph problem, for example, all those appearing as potential cost reducers at only one arc.

As a first level heuristic to minimizing the cost, we would propose minimization with respect to all transpositions τ as described above. Although this technique is probably powerful enough to handle all the problems that don't arise in practice

anyway, there are some further observations that are simply too intriguing to be omitted from this discussion. These are motivated by the following worry. Suppose that the algorithm for ordering mxcc-sets labels only one arc with a non-trivial permutation, but just happens to put it in the wrong place. Can't we use some simple technique, like the network flow analysis of the previous section, to find the right place?

Our approach is based on the following result. It shows that while we cannot consider arcs to be undirected in the general case, we have great freedom in reversing their direction.

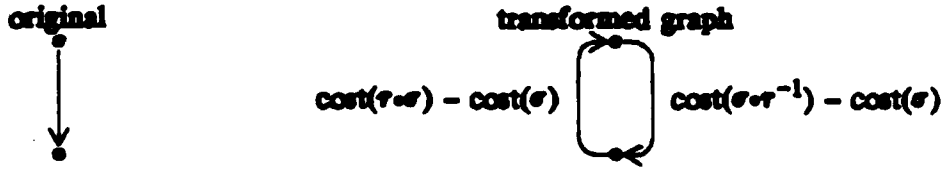
Lemma 9.5 Given a permutation-labeled graph, reverse one of its arcs, and replace the label on the arc by the inverse of the original label. The minimal equivalent labeling problem has the same solution for each graph.

Proof Look at the reversed arc. In the original graph, let σ be the label on the arc from N_1 to N_2. In the modified graph, σ^{-1} will be a label on the arc from N_2 to N_1. Now, τ-fire N_1 in each graph. In the first, the label becomes $\sigma \circ \tau^{-1}$, while in the second, the label becomes $\tau \circ \sigma^{-1}$. Since these labels are inverses, their costs are equal by Lemma 9.4. Similar remarks apply to N_2.

□

The interpretation of reversing an arc in the flowgraph, or even the mxcc-graph derived from it, is difficult to contemplate.

This lemma suggests that for τ not necessarily a transposition, we may reduce the problem of finding a subset of τ-firings that minimize the cost not to an undirected marked graph, but to a heuristic that strongly resembles the one used for undirected marked graphs (Algorithm 9.6). We replace each directed labeled arc of the original graph with a two-cycle of directed arcs, each weighted with the extra expense of τ-firing the node at entry to the arc. A negative extra expense is interpreted as a gain.



The idea of the heuristic may be summarized as follows. Let M be the set of all of the arcs in the transformed graph that have negative weight, and let M_0 be any subset

of M.

Algorithm 9.7 Try for reduced labeling with M_0

Label as a source each node N such that
  N is an entry node for some arc in M_0, and
  N is not an exit node for any arc in M_0.
Label sinks by the dual rule.
Remove all arcs in M from the transformed graph
Find a min cut

Let F be the set of nodes such that ∂F is the min-cut and such that F contains the sources. Then τ-firing the elements of F will reduce the cost of the labeling by at least

$$\text{cost}(M) - \text{cost}(\partial F)$$

Any extra reduction comes because an arc with negative weight was in the cut, but not in M. The network flow algorithm is able to take into account the different costs involved in firing different ends of an arc, given an initial choice of orientation. In Algorithm 9.6, the different orientations were tried as part of the heuristic. Here, each element of M brings its orientation with it. If M has too many elements to try all the subsets, we can use the following greedy heuristic.

Algorithm 9.8 Approximate minimal τ-firings

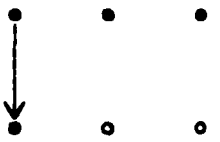
Sort M in order of decreasing |weight|
$M_0 \leftarrow \phi$
for $A \leftarrow$ each element of M
  new$M_0 \leftarrow M_0 \cup \{A\}$
  if S_0 contains arc opposite to A
    remove that arc from newM_0
  Try for reduced labeling with newM_0
  if this reduces the cost
    $M_0 \leftarrow$ newM_0

Consider the effect of this algorithm when τ is a transposition. Then the pairs of opposite arcs will have the same weight. If such arcs are next to each other after M is sorted, the above algorithm is exactly the same as Algorithm 9.6. As before, other heuristics might be proposed; we end our discussion here.

9.4 Coset Labeled Graphs

There is one final topic yet to be discussed. We have discussed only the case $H_* = H_*^n$, where permutations are the natural labels on the arcs. When traversing a boundary arc from H_*^n to H_*^m where $n > m$, a permutation is no longer the natural

label. For example, from H_*^3 to H_*^1, we might have (hollow nodes are unused variables):

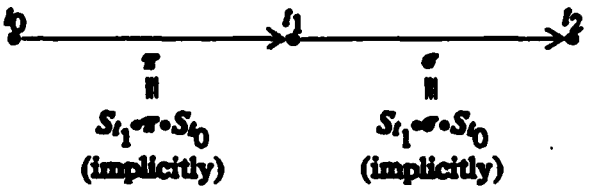


Even when traversing to a yet unseen node, it is not quite right to view this as an identity permutation on three elements, because what happens to 2 and 3 is unimportant. We could try to model this with some sort of subset selection, but it is technically more convenient to handle it group-theoretically, using the idea of cosets. We will translate "what happens to 2 and 3 is unimportant" to "the coset consisting of the identity followed by (multiplied on the left by) $S_{\{2,3\}}$", where $S_{\{2,3\}}$ denotes the subgroup of S_n that has all the permutations of 2 and 3. Let ι be the identity element of S_n. The coset is:

$$S_{\{2,3\}} \circ \iota \triangleq \{ \sigma \circ \iota \mid \sigma \in S_{\{2,3\}} \}$$

In this case, as in all cases when propagating to an unseen node, the coset is actually a subgroup of S_n because the permutation generating it may be taken to be the identity.

To say all this in proper generality, we must review the notion of type that we introduced for the $n = 2$ case (see Algorithm 9.3). There we labeled nodes of H_*^1 as type 1 or type 2. In general, we must label nodes in H_*^m with a *subset* of integers. For use in coset construction, it is convenient from the subset to specify the variables that *don't* appear; thus the type of a node in H_*^m will be a subset of $n-m$ elements. In the two variable case, type 1 we now view as type {2}. As a technical convenience, we can view the type of a node of H_*^n as ϕ; with this convention, the cosets on H_*^n will consist of single elements, and this section reduces to the previous section. We will continue to write a permutation on the arc, and construct the coset implicitly.



A graph labeled and interpreted like the above is a *coset labeled graph*. To complete the definition, we must review τ-firing a node and how to obtain a cost from a

permutation.

Suppose we τ-fire the node labeled t_1 above. First, realize that this changes the type of the node from t_1 to $\tau(t_1) \triangleq \{ \tau i \mid i \in t_1 \}$. This is clear from the original diagram for τ-firing on page 108. Once this is done, we may replace σ by $\sigma \circ \tau^{-1}$ and π by $\tau \circ \pi$, just as before. To see why, compare the products from going through both arcs.

$$(S_{t_2} \circ \sigma \circ S_{t_1}) \circ (S_{t_1} \circ \pi \circ S_{t_0}) \text{ versus } (S_{t_2} \circ \sigma \circ \tau^{-1} \circ S_{\tau(t_1)}) \circ (S_{\tau(t_1)} \circ \tau \circ \pi \circ S_{t_0})$$

To show identity we need show only that $S_{t_1} = \tau^{-1} \circ S_{\tau(t_1)} \circ \tau$, which is immediate. In summary, to τ-fire a node, we use the same rule for coset representative as we had before, and permute the type by τ.

In H_n^n, we were able to relate the cost of a permutation to its canonical representation as a product of disjoint cycles. This was a realistic reflection of the implementation of a permutation, and was convenient group-theoretically. When the types are non-null, we must pick a canonical form for cosets $S_{t_2} \circ \sigma \circ S_{t_1}$, and make sure this canonical form reflects the implementation. What freedom is there in picking another σ? The most general situation comes from looking at two (disjoint) cycles of σ, each having distinct elements $i_1, j_1 \in t_1$ and $i_2, j_2 \in t_2$. Write these cycles beginning with the element of t_2:

$$(i_2\ a_2 \ldots b_2\ i_1\ a_1 \ldots b_1)(j_2\ c_2 \ldots d_2\ j_1\ c_1 \ldots d_1)$$

Composing on the left by $(i_2\ j_2) \in S_{t_2}$ and on the right by $(i_1\ j_1) \in S_{t_1}$ gives another permutation ρ with $S_{t_2} \circ \rho \circ S_{t_1}$ equal to the original coset. This operation leaves the other cycles of σ unchanged, and has the following effect on the above cycles:

$$(i_2\ a_2 \ldots b_2\ i_1\ c_1 \ldots d_1)(j_2\ c_2 \ldots d_2\ j_1\ a_1 \ldots b_1)$$

A variant of this occurs if we have a common element j in $t_2 \cap t_1$, as if $j_1 = j_2$ and $d_1 \ldots c_1$ is null:

$$(i_2\ j)(i_2\ a_2 \ldots b_2\ i_1\ a_1 \ldots b_1)(j\ c \ldots d)(i_1\ j) = (i_2\ a_2 \ldots b_2\ i_1\ c \ldots d)(j\ a_1 \ldots b_1)$$

The pattern to notice here is that what is on the way from i_2 to i_1 or j_2 to j_1 is unchanged; what is on the way from i_1 to i_2 or j_1 to j_2 or j to j is interchangeable. There are several ways to turn this observation into a canonical form. We choose to record the t_2-to-t_1 information in the syntax $(i_2\ a_2 \ldots b_2\ i_1|$ and the t_1-to-t_2 information using $|a_1 \ldots b_1)$. Call these left and right *half-cycles*. The following result is the essence of why half-cycles give rise to a unique representation of cosets.

Lemma 9.6 Given t_1, $t_2 \subseteq \{1, \ldots, n\}$, and σ, $\rho \in S_n$; suppose $S_{t_2} \circ \sigma \circ S_{t_1} = S_{t_2} \circ \rho \circ S_{t_1}$. Let $i \notin t_1$. If $\sigma(i) \notin t_2$, then $\rho(i) = \sigma(i)$; otherwise $\rho(i) \in t_2$ also. Similarly, let $j \notin t_2$. If $\sigma^{-1}(j) \notin t_1$, then $\rho^{-1}(i) = \sigma^{-1}(i)$; otherwise $\rho^{-1}(i) \in t_1$ also.

Proof Suppose $i \notin t_1$ and $\sigma(i) \notin t_2$. Any element of $S_{t_2} \circ \sigma \circ S_{t_1}$ takes i to $\sigma(i)$. Hence, so must every element of $S_{t_2} \circ \rho \circ S_{t_1}$. Since elements of S_{t_1} do not affect i, every element of $S_{t_2} \circ \rho$ must take i to $\sigma(i)$. Assume $\rho(i) \in t_2$. Then any element of S_{t_2} will take $\rho(i)$ to another element in t_2, i.e., not to $\sigma(i)$, contradiction. Thus we must have $\rho(i) \notin t_2$, and $\rho(i)$ is unaffected by any element of S_{t_2}. Hence, $\rho(i) = \sigma(i)$. If $\sigma(i) \in t_2$, reverse the roles of σ and ρ, and argue by contradiction. The second result has the same proof.

□

We shall define a (t_1,t_2)-representative for $S_{t_2} \circ \sigma \circ S_{t_1}$ by looking at the cycles of σ. It is convenient to view one-cycles as among the cycles of σ, that is, if $\sigma(i) = i$, then (i) is a cycle of σ. The (t_1,t_2)-representative will consist of cycles and half-cycles. At each stage in the construction, we shall show that the representation is independent of the choice of σ. As a first step, it is convenient to adjust σ so that its cycles do not have repetitions of elements in t_1 or t_2.

Lemma 9.7 Given t_1, t_2, σ as above. We can find ρ such that $S_{t_2} \circ \sigma \circ S_{t_1} = S_{t_2} \circ \rho \circ S_{t_1}$ and such that every cycle of ρ has no more than one element of t_1 and no more than one element of t_2.

Proof Inductively, it is sufficient to consider one cycle at a time. If a cycle has, say, elements 1 and 2, both in t_2, compose on the left by $(1\ 2) \in S_{t_2}$. This leaves the cosets unchanged, and cuts the cycle:

$$(1\ 2)(1\ a_1 \ldots b_1\ 2\ a_2 \ldots b_2) = (1\ a_1 \ldots b_1)(2\ a_2 \ldots b_2)$$

Composing on the right has a similar effect.

□

Henceforth, we will assume that representatives for a coset meet the conditions for ρ in this lemma. A further simplifying effect is provided by the following result.

Lemma 9.8 Let $i_2 \notin t_2 - t_1$ and suppose $\sigma(i_2) = i_2$. Then

$$S_{t_2} \circ \sigma \circ S_{t_1} = S_{t_2 - \{i_2\}} \circ \sigma \circ S_{t_1}$$

There is a symmetric result for $i_1 \in t_2 - t_1$.

Proof The result follows from:

$$S_{t_2 - \{i_2\}} \circ \sigma \circ S_{t_1} = \{ \pi \in S_{t_2} \circ \sigma \circ S_{t_1} \mid \pi(i_2) = i_2 \}$$

This is because σ leaves i_2 fixed (by assumption) and S_{t_1} leaves i_2 fixed ($i_2 \notin t_1$). The symmetric result has a symmetric proof.

□

Suppose there is some $i_2 \in t_2 - t_1$ with $\sigma(i_2) = i_2$. We claim that this condition will hold for any choice of σ meeting the condition of Lemma 9.7. Let ρ be another such choice. By Lemma 9.6, $\rho(i_2) \notin t_2$, but since elements of t_2 cannot be repeated in a cycle of ρ, we must have $\rho(i_2) = i_2$. Thus, we are able to define the (t_1, t_2)-representative for $S_{t_2} \circ \sigma \circ S_{t_1}$ to be the $(t_1, t_2 - \{i_2\})$-representative for $S_{t_2} \circ \sigma' \circ S_{t_1}$, where σ' is the restriction of σ to $\{1, \ldots \hat{i}_2, \ldots n\}$. Inductively, this yields a canonical representative (the basis for the induction will appear later). We may apply a similar reduction for $i_1 \in t_1 - t_2$ when $\sigma(i_1) = i_1$.

Let us consider a cycle of σ that has no element of t_1. If the cycle has an element of t_2, let i be that element; otherwise pick i arbitrarily from the elements of the cycle. The cycle may be written in the form $(i\ \sigma(i)\ \sigma^2(i) \ldots)$. Let ρ have the same coset as σ. By Lemma 9.6, we will have $\rho^l(i) = \sigma^l(i)$ for $l <$ the length of the cycle. By the same result, ρ applied to the last element will be in t_2 if $i \in t_2$, and will equal i otherwise. Since we can adjust ρ by Lemma 9.7 so that it does not have distinct elements of t_2, we see that ρ will take the last element to i in all cases. In other words, the cycle containing i is independent of the choice of representative. Now, let σ' be the permutation of $\{1, \ldots, n\} - \{i, \sigma(i), \ldots\}$ obtained by removing the cycle from σ. We claim that $S_{t_2 - \{i\}} \circ \sigma' \circ S_{t_1}$ is independent of the choice of σ. To see this, let ρ be another representative, and form ρ' in the same way. Denote the cycle by γ.

$$S_{t_2} \circ \sigma \circ S_{t_1} = S_{t_2} \circ \rho \circ S_{t_1} \qquad \text{(by assumption)}$$

$$\Rightarrow S_{t_2} \circ \sigma' \circ \gamma \circ S_{t_1} = S_{t_2} \circ \rho' \circ \gamma \circ S_{t_1} \qquad \text{(by def. of } \sigma', \rho')$$

$$\Rightarrow S_{t_2} \circ \sigma' \circ S_{t_1} \circ \gamma = S_{t_2} \circ \rho' \circ S_{t_1} \circ \gamma \qquad (\gamma \cap t_1 = \phi)$$

$$\Rightarrow S_{t_2} \circ \sigma' \circ S_{t_1} = S_{t_2} \circ \rho' \circ S_{t_1} \qquad \text{(compose on right by } \gamma^{-1})$$

$$\Rightarrow S_{t_2 - \{i\}} \circ \sigma' \circ S_{t_1} = S_{t_2 - \{i\}} \circ \rho' \circ S_{t_1} \qquad \text{(Lemma 9.8)}$$

We define the (t_1, t_2)-representative of $S_{t_2} \circ \sigma \circ S_{t_1}$ to be:

$\{\gamma\} \cup$ the $(t_1, t_2 - \{i\})$-representative of $S_{t_2 - \{i\}} \circ \sigma \circ S_{t_1}$

Since both pieces of this definition do not depend upon the choice of σ (the latter

inductively), the definition itself does not depend upon the choice of σ.

More briefly, we consider a cycle σ that has no element of t_2. The construction is like the above, except that we use σ^{-1} to determine the rest of the cycle γ, and in the proof we bring γ out to the left, not the right, and worry about $t_1-\{i\}$ rather than $t_2-\{i\}$.

We are reduced to the case in which every cycle of σ has exactly one element of t_1 and one element of t_2. One possibility is that the elements are distinct: $i_1 \in t_1$, $i_2 \in t_2$ and $i_1 \neq i_2$. Write the cycle with i_2 first, and form the left and right half-cycles:

$$(i_2\ a \ldots b\ i_1\ c \ldots d) \mapsto (i_2\ a \ldots b\ i_1| \text{ and } |c \ldots d)$$

If "$c \ldots d$" stands for no elements, we form the empty right half-cycle $|\ )$. If "$a \ldots b$" stands for no elements, we naturally have $(i_2\ i_1|$. The only other possibility is that the cycle has an element $i \in t_1 \cap t_2$. We again get two half-cycles:

$$(i\ c \ldots d) \mapsto (\ | \text{ and } |c \ldots d)$$

Again, the right half-cycle may be empty.

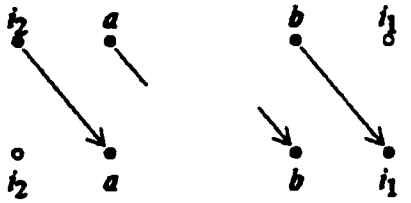
Why is this construction canonical? First, by the reduction stemming from Lemma 9.8, every element of t_2-t_1 and t_1-t_2 will appear in some cycle. By Lemma 9.6, we may start at $i_2 \in t_2-t_1$ and iterate σ until we encounter an element $i_1 \in t_1-t_2$ (other possibilities have been eliminated). Both i_1 and everything on the way from i_2 to i_1 are determined independently of the choice of σ; thus, the set of non-empty left half-cycles is independent of σ. But *every* element of $t_2 \cap t_1$ will also appear in a cycle of σ, so the *number* of empty left half-cycles is also fixed. This proves that the determination of left half-cycles is canonical.

Suppose that there is some element not in $t_1 \cup t_2$, and not on the way from i_2 to i_1 in some cycle. If we iterate σ on such an element, we eventually get to an element of t_2; if we iterate σ^{-1}, we get to an element of t_1 (maybe the same element). Such a segment of elements, not including endpoints, is independent of choice of σ; these must appear in the non-empty right half-cycles $|c \ldots d)$. But the number of right half-cycles equals the number of left half-cycles, so the remaining right half-cycles must be empty, and their number is independent of σ. This completes the proof that the set of half-cycles is unique, and provides the basis of the proof by induction of the following:

Theorem 9.7 There exists a unique (t_1,t_2)-representative for every coset of the form $S_{t_2} \circ \sigma \circ S_{t_1}$, and it may be (easily) computed, given σ.

□

We now relate (t_1,t_2)-representation to the code required to implement it, beginning with left half-cycles. The arrows in the picture below indicate the move instructions that are required:



The code sequence required is (subscripts name variables, not occurrences):

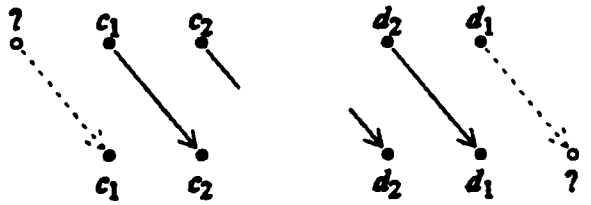
MOVE $V^*_{i_1}$, V_b

...

MOVE V^*_a, V_{i_2}

Because V_i is not used initially (this was the definition of t_1) implementing a left half-cycle is simpler than implementing a cycle—no exchanges are necessary, and there is no need to use a temporary.

The implementation of a right half-cycle is similarly easy:



Recall that the last element of a right half-cycle goes to "some element" of t_2. But by definition t_2 are the set of elements that we don't care about, so no move is actually necessary. Similarly, the first element of a right half-cycle comes from "some element" of t_1, again, a variable that we do not care about. The only moves that are necessary are:

MOVE $V^*_{d_1}$, V_{d_2}

...

MOVE $V^*_{c_2}$, V_{c_1}

In particular, if a right half-cycle has only one element, no moves are necessary.

From the discussion of the implementation, it is clear that the cost of a coset depends upon the structure of its (t_1,t_2)-representative. We extend the notion of signature:

Definition Given t_1, t_2, σ as usual, the signature of $S_{t_2} \circ \sigma \circ S_{t_1}$ is obtained from its (t_1,t_2)-representative by forming a triple of multi-sets, one each for the lengths of cycles, left half-cycles and right half-cycles.

□

In order to be able to use the techniques that we developed for permutation-labeled graphs, we want to have an invariance of cost under "conjugation"—firing both ends of an arc by τ—and inverse.

Theorem 9.8 The signature of a coset is invariant under conjugation and inverse.

Proof Let the initial coset on an arc be $S_{t_2} \circ \sigma \circ S_{t_1}$. If we τ-fire both ends of the arc, we get the coset

$$S_{\tau(t_2)} \circ \tau \circ \sigma \circ \tau^{-1} \circ S_{\tau(t_1)}$$

Express σ as a product of cycles. The effect of $\tau \circ \sigma \circ \tau^{-1}$ is to replace each element i in the representation by τi (this is the essence of the proof of Lemma 9.4). The calculation of the (t_1,t_2)-representation for σ proceeded by looking only at t_1, t_2 and the cycles of σ. If these elements are permuted by τ, the final result will have elements permuted by τ.

To prove the result for inverse, first note:

$$(S_{t_2} \circ \sigma \circ S_{t_1})^{-1} = S_{t_1}{}^{-1} \circ \sigma^{-1} \circ S_{t_2}{}^{-1} = S_{t_1} \circ \sigma^{-1} \circ S_{t_2}$$

The cycles and half-cycles in a (t_1,t_2)-representative are simply reversed to obtain those for a (t_2,t_1)-representative of the inverse:

$$(a \ldots b) \mapsto (b \ldots a)$$

$$(t_2\ a \ldots b\ t_1| \mapsto (t_1\ b \ldots a\ t_2|$$

$$|c \ldots d) \mapsto |d \ldots c)$$

A detailed proof that this works is omitted.

□

The techniques that we developed for permutation-labeled graphs depended on very little:

- invariance under conjugation and arc-reversal
- comparison of cost(σ) versus cost($\sigma \circ \tau$)

We have demonstrated how to do both of these. Thus, the techniques of the previous section apply to coset labeled graphs, rather than just permutation labeled graphs.

References

1. Ford, L.R., and Fulkerson, D.R. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

2. Hall, Marshall, Jr. *The Theory of Groups.* Macmillan, New York, 1959.

3. Lewis, H. and Papadimitriou, C. The Efficiency of Algorithms. *Sci. Am. 238* (January 1978), 96 ff.

4. Matula, D.W., and Beck L.L. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM 30* (1983), 417-427.

5. Tarjan, Robert E. Depth first search and linear graph algorithms. *SIAM J. Computing 1* (1972), 146-160.

6. Tarjan, Robert E. Solving Path Problems On Directed Graphs. STAN-CS-75-528, Stanford University, November, 1975.

7. Tarjan, R.E. Efficiency of a good but not linear set union algorithm. *J. ACM 22* (1975), 215-225.

8. Wulf, William, et al. *The Design of an Optimizing Compiler.* Elsevier, New York, 1975.

Index

An Intermediate Form for Bi-directional Scanning of Programs

1. Introduction

The purpose of this document is to discuss a data structure that aids in the bi-directional scanning of programs. Our primary motivation for bi-directional scanning is in the live-dead analysis that aids in the register allocation aspects of code-generation; it is also useful in some aspects of program verification, and proofs of program termination.

The data structure that we use may be based on any representation for a directed graph that allows arcs to be traversed in either direction. We assume that the reader is acquainted with the basic operations on directed graphs. The source language that we use here will be based on tree like representations of functions, not unlike those found in LISP or EL1. The generality required to handle these languages is more than adequate to handle languages such as FORTRAN or PASCAL.

2. Getting started

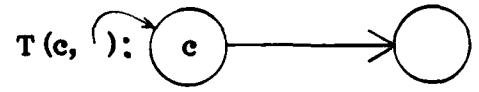
We shall take as our source language a language based on the lambda calculus, but with declarations added. This gives us a very simple syntax. A term is one of the following:

constant	
variable	
form ($term_1$, ..., term)	(application)
$\lambda x_1, \ldots, x_n$. term	(abstraction)
δx_1: term,..., x_n:term . term	(declaration)

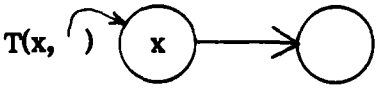
The terminal classes constant and variable will be left undefined (x_i is a variable). The only other terminals are λ and δ; these are neither variables nor constants.

The first step in constructing a flow graph will be a naive translation from the above syntactic classes to "straight-line code". This is accomplished by a function denoted by T, and having two arguments. The first is the term to be translated, and the second is the node at which scanning is to start. This node will have no contents at entry to T, but will be incorporated in existing graph structure. The result of T is the node of which scanning is to resume.

For variables and constants, T installs its first argument as the contents of its second argument, and connects the second argument to a new node, which then becomes the result of T.



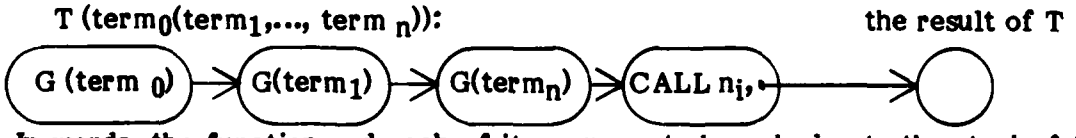
the 2nd argument of T the result of T



similarly

Implicit in these constructions is that the first node pushes the value of c or the present value of x onto a stack, and that control flows on to the next node in the scan.

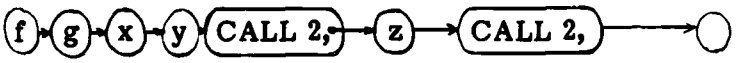
To produce the graph for an application, the graphs for the constitutents are chained. The following a graph expresses the semantics that applications are evaluated by evaluating the operator and operands from left to right, followed by the function call:



In words, the function and each of its arguments is pushed onto the stack, followed by a "call". The pseudo-op CALL has two operands. The first is n, the number of arguments to the function. The second is a pointer to a new node that becomes the result of T. There is <u>not</u> an ordinary one from the CALL node to the result of T because ordinary arcs indicate direct flow. However, the new node is the appropriate point to resume the scan.

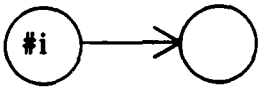
We shall later need a picture of the stack at entry to an abstraction. For purposes of specificity, we shall assume that CALL replaces the function that is n entries below the top of the stack with the return node, i.e., the second operand, just before entering the abstractions.

As an example of the graph structure set up by T, consider the term f(g(x,y),z). Its translation is:

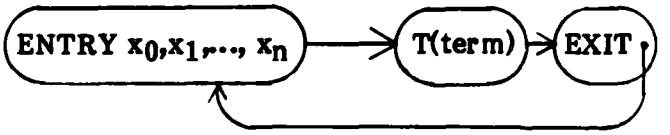


Not unlike reverse polish notation.

The graph of an abstraction involves a slight twist. On the one hand, the graph of an abstraction must involve the graph of its body (the "term" in $x_1, ..x_n$. term); on the other, in this language, an abstraction is a value. To denote the value aspect of abstractions, we will use an index number i, and treat the abstraction essentially as a constant.

T($x_1...x_n$. term) = 

The abstraction also has a program aspect, for which we want a graph. We let G i be the flowgraph for abstraction i. This graph consists mainly of a graph for term, with a header and trailer corresponding to function entry and exit.

G [i] = 

The pseudo-op ENTRY gives the names x_0, x_1, ...x_n to the top n+1 entries on the stack (x_0 names the return address preceding the first argument; how that name is chosen is not discussed here). The pseudo-op EXIT indicates that the 2^{nd} through $n+2^{nd}$ entries (return address and n arguments) are to be removed from the stack, leaving the top entry as the result of the call. EXIT's operand is a back pointer to the entry node of the abstraction, a useful piece of information, as we shall see.

The graph of a declaration involves the graphs of $term_1$, ...$term_n$ (which

provide the initial values for the x_1), as well as the graph for $term_0$

$T(x_1: term_1 \ldots\ldots x_n: term_n \cdot term_0) =$

The pseudo-op SCOPE indicates entry to a scope in which mutually recursive functions can be declared, DECL indicated the point at which the names take effect. The pseudo-op UNDECL has single operand indicating the DECL that is to be undone. The 2nd through n+1st entries are removed, leaving the result of $term_0$ as the result of the declaration.

3. Weak Interpretation

The purpose of this section is to describe how to splice together the various graph fragments that have been constructed so that flow of control on function calls is represented by arcs in the graph. Because functions are values, the process of constructing this full graph must involve essentially a "symbolic execution" or "weak interpretation" of the program. The technique described here has more the flavor of weak interpretation than of symbolic execution, insofar as there is a real difference.

We shall provide a property set [1] or monotone framework [2] for modeling the state of an actual execution. Since the state for the language just described is entirely contained in the stack, we will call the particular property set that is developed below a stack model, or simply sm.

Every node of the graph will have an associated sm. Each sm may be relatively large object, and it may appear that the amount of data is massive. However, even though at one level we view each sm as distinct, at another level we provide for a great amount of sharing between sm's at adjacent nodes of the flowgraph. The sm field at each node is of constant size - a pointer - and we expect the total amount of space consumed by sm's to grow roughly linearly with the size of the program.

At any time in the execution of a program, the stack is a linear sequence of values (for uniformity, we think of return addresses as values), which we may view as a sequence of values with the top of the stack at the beginning of the sequence . In a recursive program, the stack may grow arbitrarily large, and so a stack model cannot directly reflect all the entries in all possible stacks. Instead, a stack model is a graph with a distinguished node, called the top node, because it corresponds to the top of the stack (this graph is not to be confused with the flowgraph). The sm field of a flowgraph node points to the top node. The sense in which an sm models an actual stack is that any actual stack will be a path through the graph that constitutes the sm. But the sm is of bounded size (as we shall see), and will thus have cycles in a recursive program. It may be the case that there are paths through an sm that do not correspond to possible stacks during execution. These represent loss of information by the modeling process, a necessary property of any analysis program that always terminates.

Running through the stack is the static chain, which allows finding the stack locations of names that are visible at that point in the execution. To model the static chain, certain nodes of an sm will be marked as static-chain nodes. These nodes will have a list of associated names, a pointer to the sm node corresponding to the last of these names, and a pointer to the next node in the static chain. Because the static chain is bounded in an execution, the model of the static chain reflects the actual static chain precisely.

In order to splice together graph fragments, sm must keep track of two types of value, functions and return addresses. To keep track of functions, we attach to each sm node a set of functions, representing the set of all functions that are possible for the value corresponding to the sm node. There are two types of functions. One is a constant function, which we may think of as a built-in. The

other kind of function is a pair i,s where i is the index of an abstraction and s is the static chain in which that abstraction is to be evaluated (an abstraction alone is not a function; it must be supplied with an environment).

To keep track of return addresses, we do not label the sm node corresponding to the return address; rather, we label the sm arcs arriving at such a node, each one having a distinct label. As we shall see, this provides enough information to model execution reasonably accurately.

4. Node operations on stack models

In this section we will describe the weak interpretation of flowgraph nodes. Each flowgraph node has an associated sm describing the state prior to execution of that node; we show how to combine the prior sm and the contents of the node to yield a set of sm/flowgraph node pairs that describe the possible states after execution. The action to be taken with this set will be described in the next section.

Weak interpretation begins at an abstraction designated as the "main program". Assume that this abstraction has n arguments. As we shall see in the next section, its ENTRY node will point to an sm that is a simple chain of n +1 nodes. The last node in the chain corresponds to the return address. The fact that this sm node has no arcs leaving it will indicate to the weak interpreter that exit from this abstraction corresponds to program exit. In order to begin weak interpretation at any abstraction, not just the first, it is necessary to have not only the sm corresponding to the state upon entry, but also a model of the static chain, which we denote by s. In situations where there are no symbols defined outside the program, s may be nil; if there are symbols that must be known during weak interpretation, these can be represented by s. To process an ENTRY node, the weak interpreter "adjoins" a node to the prior sm. This means that a new node is

obtained, and an arc is established from the new node to the prior sm, i.e., to the top node of that sm. (As we shall see, this basic operation is used in the weak interpretation of several node types; it corresponds to a "push" on to the interpreter's stack). In the case of an ENTRY node the adjoined node becomes a static chain node. The list of names is the operand to ENTRY. The sm node corresponding to the last of these names is the top node of the prior sm (corresponding to the last argument). The static chain link for this static chain node is of course s. In this case, as in later ones, the adjoined node becomes the top node of the new sm. The result of the weak interpretation of the entry node is a singleton set whose pair is the new sm just described, and the flowgraph node pointed to by the (necessarily) unique are out of the ENTRY node.

The weak interpretation of a node containing a constant also involves adjoining a new node to the prior sm. It is necessary to attach the set of functions that this constant will evaluate to. If the constant is not a function, this set is null. If the constant is a function, then the set is a singleton consisting of the constant.

A node containing a variable is processed in the same way as a constant node, once the function set is obtained. To obtain the function set, we "look up" the variable in much the same way that a value interpreter would. Specifically, beginning at the node pointed to be the sm field, follow sm arcs until a static-chain node is seen. If the desired variable is not among those at this static level, then follow the link to the next sm node in the static chain, and repeat the above step at that node. If the static-chain pointer is nil, the variable is undefined and the program has an error. If the desired variable is in the set of names at a node, then we can find its offset from the last of the names, and we also know the sm node n corresponding to the last of the names. Call the offset p; if we traverse p sm arcs from n, we arrive at a node corresponding to the variable being looked up. The

only sm nodes with multiple out arcs correspond to return addresses, and these always occur as the first element in a list of names. Thus there is never any choice in traversing the p arcs, and the name specifies a unique sm node.

A node corresponding to an abstraction (pseudo op #) is also treated like a constant or variable node, once the function set is found. In this case, the function set is a singleton, consisting of the pair i,s, where s is the first static-node seen when folowing sm arcs from the node pointed to be the sm field.

For constant, variable and abstraction nodes, the result of weak interpretation is a singleton set, whose pair is the described sm, and the node obtained by following the unique flowgraph are out of the node.

The processing of the CALL and EXIT pseudo-ops provide the interprocedural linking that we desire. We consider CALL first. The first operand (the n) and the sm field attached to the node yield the function set that is possible from the call. The result of weak interpretation will have one sm/node pair for every element of the function set.

We first examine built-in functions. Each such function has built-in semantics that must be properly represented by the graph. Many built-in functions have little interaction with flow of control. For instance, none of the arguments of + are functions; its result is not a function and its effect is merely to go on to the next step. This effect can be represented by connecting the CALL node to the second operand of the CALL indicating the flow of control that actually occurs. The sm for such a function is obtained by traversing n in arcs from the node pointed to by the sm field of the call node (corresponding to popping the arguments and the function value), and then adjoining at this sm node a node that represents the result of the built-in (corresponding to the push). The pair for such a function is the sm just described, and the second operand of the CALL.

There are some built-in functions that are more interesting: those that affect flow of control, and assignment. These are discussed in later sections.

To process a pair i,s corresponding to an abstraction, we imitate in the sm the action that would be taken in execution, where the function call is replaced with the return address. This cannot be done directly in the sm because of data sharing. Instead, it is necessary adjoin a new node to the node n+1 steps from the top of the prior sm. This node corresponds to the return address and the new sm arc is labeled with the return node. Then a copy is made of the top n nodes of the prior sm, and the last of these nodes is linked to the return address node that was just adjoined. Recall that beginning the weak interpretation of an abstraction requires not only sm, but also the static chain. In this case, that is simply the second component of the pair i,s. The flow of control from the CALL node node to the ENTRY node is indicted by establishing a flowgraph arc from one to the other. The contribution that an abstraction makes to the result of weak interpretation is the pair consisting of the described sm and the ENTRY node of the abstraction.

The sm associated with node containing an EXIT pseudo-op has a special form. The top node represents the result of the application. Traversing one sm arc arrives at a static-chain node, which tells how many more steps mst be traversed to arrive at the node corresponding to the return address. The result of the weak interpretation of the EXIT node will have one pair for each arc leaving the return address node, where the node component for the pair is the label on the arc (the return node). Copy the top of the prior sm and make it point to the node at the other end of the labeled sm arc, representing the removal of the return address and arguments. The new node is the top node of the sm associated with the return (flowgraph) node. To indicate flow of control, establish an arc from the EXIT node

to the return node.

The pseudo-ops remaining to be discussed are all associated with declaration: SCOPE, DECL and UNDECL. The actions of these pseudo-ops must be coordinated, but there are several ways of achieving the desired effect: allowing the definition of a mutually recursive set of routines. We shall present one method here, corresponding rather transparently to a reasonable implementation. To preview: SCOPE establishes a new static chain link, but with no name; DECL fills in the names and the stack pointer; and UNDECL removes everything except the top of the stack. We now discuss each case in more detail.

To weakly interpret SCOPE, adjoin a new node to the prior sm. This node will be a static chain node, whose static chain link is obtained by traversing the prior sm until a static chain link is found (there will be no branching along this sm path). The list of names for this static chain node, as we stated above, is empty. Since no names will be found at this static chain link under these conditions, the pointer to the sm node for the last name is irrelevant, and may as well be nil. Note that this allows the weak interpretation (and evaluation) of $term_1$, ..., $term_n$ in an environment where the names have not yet been installed.

To weakly interpret DECL, traverse n sm arcs from the distinguished node of the prior sm (there will be no branching along this path), arriving at the static chain node that was established by SCOPE. The effect of DECL is merely to install into the name field here the list of names of the DECL, and to set the previously nil pointer to the present distinguished sm node, i.e. the top of stack. This cannot be done to the extant sm graph structure, because that would invalidate the stack models of flowgraph nodes pointing to the shared structure. It is necessary, at least conceptually, to copy the n+1 nodes that represent the top of the stack, and only then make the changes to the static chain node. The first node of the copied chain is the top node for the new sm.

The weak interpretation of UNDECL is what one would expect: the top sm node is copied, but attached to the node obtained by traversing n+1 arcs past it, thereby popping not only the n variables, but also restoring the static chain to what it was before the declaration.

In the processing of every node type, observe that the amount of new (unshared) graph structure is proportional to the size of the program. In fact, for all but DECLs, only one new sm node is required at each point. For a DECL, the number of nodes is the number of variables plus one, but variables contribute to program length.

5. Disjunction

Weak interpretation uses a set Q of "unprocessed nodes". These are nodes that must be processed before a consistent weak interpretation has been attained. We have already noted that at the beginning of weak interpretation the ENTRY node of the main program is given an initial sm. The rest of initialization consists of ensuring that the sm fields of all other nodes are set to nil (meaning that flow cannot arrive at this node), and initializing Q to the singleton whose element is the entry node.

The general outline of weak interpretation consists of removing a node from Q, and processing it according to the description of the previous section. The result is a set of pairs (sm, fn) consisting of an sm that is valid prior to the flowgraph node fn. The key to termination of weak interpretation is whether fn is placed back in Q, and if so, what is the value of the new sm field of fn. The theory of weak interpretation says that the key to obtaining a correct and terminating weak interpretation is the definition of a suitable disjunction operation on stack models. Given such a disjunction operation, we apply it to sm and the sm already

attached to fn. If the result is the same as the old sm attached to fn, fn is not placed back in Q. Otherwise, fn is put back in Q, and its new sm is the result of disjunction.

We now consider the disjunction of sm_1 and sm_2. If sm_1 is nil, then the result is sm_2; if sm_2 is nil, the result is sm_1. This corresponds to the observation that if the sm field of n node is nil, then flow does not arrive there. The logical or of this condition and any logically weaker condition (flow arrives here with the stack having thus and such a shape) is the logically weaker condition. Since a nil sm is never propagated forward, if this case applies, the node is always put back in Q. If weak interpretation terminates and the sm field of some node is nil, then it is indeed true that flow never arrives at that node.

If sm_1 and sm_2 one both non-nil, we have a more interesting question. To see how to define disjunction, recall that the basic definition of a stack model is that its paths limit what might be seen as the values of a stack. Suppose we want to construct sm_1 whose set of paths is as small as possible, but still contains the union of the paths for sm_1 and sm_2. Ignoring efficiency considerations for the moment, let sm_0 be the graph consisting of disjoint copies of sm_1 and sm_2. We will describe a process called "pinching". This begins at the top nodes of sm_1 and sm_2. To pinch two nodes, coalesce them in the graph, and attach as a function set the union of the function sets. Then examine the out arcs of the two nodes. If any of the out arcs are labeled, then all must be labeled, because we have just pinched a return address node. In this case, we pinch nodes at the ends of identically labeled arcs (which come in pairs). Otherwise, there will be at most a pair of out arcs (corresponding to the single out arcs in sm_1 and sm_2). If there is a pair, pinch the two nodes.

While this is not the place for a detailed proof, completion of pinching results in a stack model having the desired validity property, in other words, the union of

the set of paths of sm_1 and sm_2 is contained in the set of sm_0, and this is the smallest set of paths describable by a stack model. More interesting is the fact that even though the semilattice described by this disjunction operation is not well-founded, weak interpretation still terminates. To prove this, we observe that an sm-node for the return address of a particular abstraction can appear at most once in any sm attached to a node. This is certainly true initially. The only place that a return address sm node is adjoined is at a CALL node. This may temporarily produce an sm with two return nodes for a particular abstraction, but in the disjunction that necessarily precedes attachment to a flowgraph, the two nodes are merged (this is how cycles arise in the sm graph). To the observation that the number of return address nodes is finite, we add the observation that the out degree of these nodes is bounded, since each such sm arc is labeled by a distinct flowgraph node, and there are a fixed number of these. The paths between return nodes cannot grow indefinitely, and thus the sm graph is finite.

6. Assignment

In confronting the issue of assignment, the issues of parameter passing, aliasing, and a host of related concerns arise. We shall provide a quite simple way of viewing assignment, in which all of the other issues can be expressed. Simply put, locations will be treated as bona fide values, and locations will be kept track of in much the same way as function values. In what we have described thus far, parameter passing has been by value, and that will continue to be our model. To do parameter passing by name, simly pass locations along; aliasing is represented by a location set.

We have already seen that an abstraction is sometimes treated as a value and sometimes as a procedure. Similarly a name x is sometimes treated as a variable (as we have done already), and sometimes needs to be treated as a location (for

purposes of assignment). In the syntax for a term, we have discussed the terminal class "variables"; symbols in this class are treated as variables, i.e., evaluating them yields their value. Without raising the issue of surface syntax, it is possible for constant to be a symbol. In this case, the evaluation of the constant is the location on the stack designated by the symbol at the time of evaluation.

In order to model assignment, we must extend the notion of a stack model, in that now, we associate not only a function set with each node, but also a location set. (We can arrange the representation so that the representation of two null sets is no more expensive than was the representation of the single null set. If the language is strongly typed, we might also take advantage of the fact that the two sets cannot be simultaneously non-null.) The representation of a location is a pointer to an sm node.

It is necessary to describe how to weakly interpret a constant node that is a symbol. Given a prior sm, the symbol is "looked up" in the same way as a variable, but the pointer to the sm node corresponding to the symbol is returned, not the attached function set. The node adjoined to the prior sm is given a singleton location set, consisting of the sm node that was found on look up. The attached function set is of course null.

In the weak interpretation of a variable node, looking up the variable now produces both a function set and a location set. These constitute the pair that are attached to the adjoined node.

We now describe the weak interpretation of the constant function ASSIGN. Its first argument produces a location, and the second, a value that will be stored in the location.

First, since ASSIGN does not affect flow of control, establish a flowgraph arc from

the CALL node to its successor. To obtain an sm for the location, we obtain a new sm for each element of the location set of the second entry in the prior sm. These sm's are disjoined by the technique of the previous section, and the result is the new sm. In most cases, there will be only one element of the location set, so there is no necessity to do a disjunction.

The problem thus reduces to describing the effect on sm of assigning to a single location. Since assignment of location values and function values is rare in typical programs, the location and function sets of both the location being assigned to, and the top of the prior sm, are most probably null. In this case, the assignment has no effect on data of interest to a stack model, and the new sm is obtained by popping three nodes off the prior sm, and adjoining a node that represents the result of assignment (depending on ones taste, this might be the first argument, the second, or a canonical nothing result).

Suppose that the assignment is of a function or location value. What we want to do is simply to change the function set location set pair to be the pair on the top of the prior sm. This cannot be done literally, because the prior sm is pointed to from the sm fields of other nodes. A correct algorithm would be to apply the prior sm before changing a field in it. A more sophisticated approach is to copy only as much as necessary, and to combine the necessary copying with the disjunction that may be necessary. These are details beneath the level of the current discussion.

Another natural operation on locations is DEREF, which we take to be a constant function. This function has a single operand which must be a location; its result is the current value of that location. The weak interpretation of DEREF is essentially like that of +; the only difference is in how the function and location sets are computed. From each location in the location set, obtain the function and location sets attached to that node; the union of all of those is attached to the top

of the new sm (a node adjoined two sm arcs from the top of the prior sm).

We thus see that accounting for assignment during weak interpretation raises no fundamental difficulties. The simple technical device of keeping track of locations, which after all is ultimately what happens during an actual execution, is sufficient to determine what control paths are possible. The method handles naturally even the assignment of function and location values, capabilities not allowed in many languages. The price is extra expense, but it is paid only when the capability is actually used.

7. Flow of Control

The purpose of this section is to show how flow of control other than function call and return fits into the process of graph construction by weak interpretation. In all cases, the flow of control will reflect the semantics of some built in function. The main point is to show that flow of control can be handled with very little mechanism over that already presented, not to give the most direct translation of the standard constructs.

We first consider how a simple if-then-else fits into this scheme. As is customary, we will assume that only one of the alternatives is evaluated depending upon the condition. In order to model this in our language, we posit a constant (built-in) function IF of three arguments, all evaluated. Either the second or third argument is the result, depending upon the first. Thus, the construct "if $term_1$ then $term_2$ else $term_3$" would be represented as (IF ($term_1$, $term_2$, $term_3$))(), and the graph structure would be:

IF G($term_1$) #2 #3 CALL 3, CALL 0,

To give a weak interpretation for the IF function is to describe its effect on a stack model and on the graph structure. Since the result can be either that of the top of the stack or next to the top of the stack, we obtain the function set for the

top entry of the new sm by taking the union of the function sets for the two entries on the top of the stack. Traverse four sm arcs from the top node (three arguments plus the function), adjoin a new sm node to this point, and attach the new function set. Then construct an arc from the CALL 3 node to a new flowgraph node, and on to the new CALL 0 node. The pair consisting of the new sm and new flowgraph node is the result of interpreting the constant function IF.

We take a similar approach to looping. We map the construct "repeat term end" to REPEAT (λ.term), where REPEAT is a constant, i.e., built-in, function. The graph becomes:

REPEAT # i CALL 1,

the semantics of the REPEAT operator is to repeatedly call its argument and throw away the result. Thus, we connect the CALL node to the following graph construct.

COPY CALL 0, POP

Here, COPY means to push the top of the stack onto the stack, and POP means to simply remove the top of the stack. The effect of these pseudo-ops on a stack model may be supplied by the reader.

In the REPEAT example, note that the flowgraph node that is the second argument to CALL 1, is never reached. This naturally raises the question, what about loops that do terminate? We answer this question by asking a different one: how can we model escape-like constructs? Our answer is to give the semantics and weak interpretation for a very general version of escape. We shall make an escape point a "first class object", just like a function or location. In fact a return point is exactly the location of a return address. While return points may be passed as arguments (or assigned), their ultimate destination is the first argument of the constant function ESCAPE. The second argument of this function is the value to

be returned as the value of the function being escaped. The effect is that everything in the stack between the top and the static chain link just above the return point (exclusive), is removed, and the next step is the same as that of the pseudo-op EXIT.

The mechanism for location sets discussed in the last section is used without change when weakly interpreting ESCAPE, which is handled as one would expect. The top node of the prior sm corresponds to the value being returned on behalf of the abstraction. The second sm node will have a non-null location set and each location will be that of a return address (if either condition is not true, there is an error). Although it is not necessary, it is convenient to have all escapes from an abstraction collect at the exit node for the abstraction. To aid in this, it is helpful to have an index of the abstraction in the top node of the sm attached to its entry node (the return address), and a way to get from abstraction index to exit node.

For each location in the location set attached to the second node of the prior sm, locate the EXIT node for the corresponding abstraction, and establish a flowgraph arc from the CALL node invoking the ESCAPE to the EXIT node. Copy the top node of the prior sm, but make it point to the static-chain just above the escape location (via the abstraction index). The set of pairs, consisting of sm's obtained in this way and the corresponding EXIT node of the abstraction, is the result of the weak interpretation of the constant function ESCAPE.

We now return to the issue of looping constructs with exits. For example, consider the standard "while $term_1$ do $term_2$. This can be viewed in our language as

WHILE($term_1$, $term_2$)

But WHILE does not need to be built in because it can be defined as follows:

condition, body. repeat if condition () then exit function else body () endif

endrepeat

The repeat and if-then-else constructs have been discussed; the end function maps directly to an ESCAPE with suitable argument.

In summary, the flow of control constructs found in most languages can be included in this general scheme with little difficulty. The "if-then-else" constructs require only a new constant function and new pseudo-ops. The "repeat" construct requires a constant function and two simple pseudo-ops for manipulating the top of the stack. The escape mechanism requires an extra abstraction in the stack model and a table of exit nodes, as well as a constant function (and no new pseudo-ops), but it provides a powerful capability. Together with ordinary function cell and procedure parameters, other control constructs are easily described.

Bibliography

1. Wegbreit, Ben. "Property Extraction in Well-founded Property Sets", Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts and Computer Science Division, Bolt, Beranek and Newman, Inc., Cambridge, Massachusetts, February, 1973.

2. Kam, J.B. and Ullman, J.D. "Monotone data flow analysis frameworks" Acta Informatica 7:3, 1977. pp 305-318.

The RULOG Inferencing Engine

Thomas E. Cheatham, Jr.
Harvard University
and
Software Options, Inc.
Cambridge, MA 02138

February 14, 1985

1 Introduction

The development of sophisticated expert systems depends upon obtaining the knowledge of experts and developing efficient means for representing that knowledge and drawing inferences from it. Some of the more successful expert systems approach the efficiency issue by computationally tractable models specific to very narrow domains with specializations of general purpose problem-solving methods applicable to a wide variety of problem domains. The general purpose component used in many expert systems is rule-based, with production rules representing knowledge and the associated mechanisms for drawing inferences.

In other systems it has seemed very natural to use the predicate calculus to represent knowledge. However, until recently, the inferencing algorithms for pure predicate calculus were too inefficient for this approach to be a practical alternative to the less formal rule-based approach. Additionally, if predicate calculus knowledge inferencing is done using the resolution method of theorem proving, it is difficult to explain conclusions or even inferences in terms that make sense to a user. Recently, the emergence of logic programming in general and PROLOG in particular suggests that

there may practical realizations of inferencing engines for predicate calculus. The theory of logic programming imposes a syntactic restriction on formulas in the predicate calculus, limiting the formulas to what are called definite clauses. The resulting language has proved adequate for a wide range of applications, and there are well understood techniques for constructing an interpreter (that is, an inferencing engine) for definite clauses that is reasonably efficient. Also, with logic programming the usual representation of the proof of some predicate is a "proof tree" that provides a very natural framework for explaining why a particular proof was successful as well as exploring why some attempted step in a proof was not successful.

Because logic programs can be given a precise semantics, they are amenable to theoretical analysis. It is even possible to attach uncertainties to the rules in logic programs while retaining precise semantics [Shapiro 83].

PROLOG is presently the best known and most widely available language for logic programming and PROLOG implementations exist on systems ranging from mainframes to personal computers. The Japanese fifth generation project has, of course, made a strong commitment to PROLOG as the basis for expert systems of the future [Feigenbaum 83], and a number of groups are investigating specialized architectures for PROLOG machines.

Yet no matter how capacious or fast the underlying hardware, we cannot ignore the importance of the efficiency of the software. Our interest is in extensions to the inferencing technology – to the software technology – to improve the performance of inferencing engines.

There are several factors that will improve the inferencing capabilities of interpreters for logic programs. One improvement is to incorporate special-purpose inferencing components for specific domains. For example, existing interpreters can infer, from the facts $x < y$ and $y < x$, that x and y are mutually inconsistent only when x and y have constant values that can be compared. To handle the general case would require axiomatizing the less-than predicate; such an approach is impractical. There are, however, efficient satisfiability procedures for systems of linear inequalities [Nelson 81]. Another example is sets of equalities and disequalities. Again, to handle general equalities and disequalities with present-day interpreters, we must axiomatize these concepts, inducing computationally infeasible

iterpretations; and, again, there do exist efficient decision procedures for handling equalities and disequalities [Nelson 79].

Another major limitation of the current generation of logic program interpreters is in their ability to deal with large numbers of ground facts. The difficulty is not just in being able to store large numbers of facts but, rather, in managing them, ensuring that they are up to date and consistent.

A third limitation of current logic program interpreters is that they lack a type system. The same argument can be made for logic programs that is made for other kinds of programs – that incorporating a type system becomes essential when the knowledgebase becomes sufficiently large.

We have recently developed an interpreter for a language based on definite clauses that has the power and generality of current logic programming interpreters but does not suffer the limitations cited above. Aspects of this language and its interpreter will be described in the sections following. In order to distinguish our rule language we call it RULOG (for RULes in LOGic). The term RULOG is also intended to suggest that our intended application is knowledge representation and inferencing, and not programming as is the case with PROLOG.

2 The RULOG Type System

It is, of course, possible to simulate a type system within PROLOG; [Mishra] describes one way of imposing types. However we have chosen to include a type system imbedded directly within the RULOG language. The type system we have chosen for RULOG is modeled on in the Ada type system for scalars extended to permit the definition of functional types. For example the following are type definitions in RULOG.

- type COLOR is (RED, BLUE, YELLOW, GREEN, MAUVE)
- type STOP-LIGHT is (RED, YELLOW, GREEN)
- type state is (odd, even)
- type small is new INTEGER range 1..100
- subtype little is small range 2..10

- type arith is function (INTEGER, INTEGER) return INTEGER
- type rel is function (INTEGER, INTEGER) return BOOLEAN

The type COLOR is an enumeration type with five distinct values named by the literals RED, ... , MAUVE. As with Ada, enumeration literals may be "overloaded" (as are RED, YELLOW, and GREEN above) and the type ambiguity is resolved by conversion, as in

...COLOR(RED)...

or by context, as in

subtype NOGO is STOP-LIGHT range RED...YELLOW

RULOG goes beyond Ada in permitting the definition of functional types; the type arith above is that of a function taking two INTEGER arguments and returning an INTEGER result. A predicate is a function returning the (built-in) type BOOLEAN.

Subtypes are, as in Ada, constraints on an underlying type. Thus, referring to the above examples, an object with subtype little has type small and is constrained to have a value between 2 and 10.

3 The RULOG Language

We can think of RULOG as having three kinds of statements: definitions, assertions, and dialogue.

Definitions

Several sorts of things can be defined in RULOG:

type and subtypes

Commencing with the built-in types INTEGER, STRING, and BOOLEAN, one can define (name) new types as enumerations and as derived from an existing type, possibly including a constraint on the parent type. Subtypes are defined as constraints on some existing type. The identifiers used to name types and subtypes may not be used for any other purpose.

literals

A literal in RULOG is like a variable in a programming language (the term variable, however, being reserved in RULOG to mean a quantified variable in some rule). An example of a literal definition is

let S: state initially odd

that defines S to be a literal with type state and initial value odd.

tokens

A token names a scalar, function, or predicate whose value is entirely determined by subsequent assertions. Some example are

- token x is a INTEGER
- token s is a function(INTEGER) returns INTEGER
- token p isa function(INTEGER, INTEGER,) returns BOOLEAN

defining x as an INTEGER, s as a function on INTEGERs to INTEGERs and p a predicate on pairs of INTEGERs.

database predicates

A database predicate defines a mapping between predicate symbols in RULOG assertions and tuples in a relational database. As an example

define has_age(name:STRING, age: small)
on employees of DB

introduces the predicate symbols "has_age". This predicate is to be taken as true for each of the two element subtuples of each tuple of the "employees" relation of the database named "DB" consisting of the value of the attribute named "name" and that of the attrribute named "age". The types of the name and age components are, within RULOG, understood to be STRING and small, respectively. The transactions betwen RULOG and the database access mechanisms will be discussed below.

Assertions

Once we have defined a sufficient collection of types, tokens, function symbols, and predicate symbols we can make assertions to RULOG. Assertions take two basic forms: assertions about uninterpreted predicates and assertions about tokens. The assertions about uninterpreted predicates are, semantically, similar to the rules in PROLOG; an example is

assert forall(e:STRING, a:small) young(e) if
has_age(e,a) and a $\leq$ 25

stating that a "young" employee is one whose age does not exceet 25. In general a rule has a forall(...) prefix that names the quantified variables and gives their types, followed by a conclusion and a conjunction of premisses. As with PROLOG, the rule asserts that the conclusion can be established by proving that each of the premisses is true. Each premiss is a predicate applied to zero or more terms; terms includes literals, tokens naming scalars, quantified variables, and tokens naming functions that are applied to terms, recursively. The major differences from ordinary logic programming rules are that the quanitified variables are typed, that the

interpretation of various of the built-in predicates like $=, \neq$, and $\leq$ is handled somewhat differently, and that certain of the predicates – the database predicates – require communication with a database system in order to be established. We will discuss the interpretation of the built-in predicates and database predicates below.

The assertions about tokens involve the equality, disequality, and inequality predicates. An example, assuming that x, y, and z had been defined as INTEGER valued tokens, is:

assert $x \leq y$
assert $y \leq z$
assert $z \leq x$

Subsequent to receiving these assertions RULOG would know that whatever value x, y, and z have they all have the same values and, for example, it would recognize that the assertion

assert $x \neq y$

was not valid (technically, it is unsatisfiable).

Dialogue

One kind of client for RULOG will be some process that wishes to determine whether or not some predicate can be proved; an example of a request for a proof would be

prove exists(E:STRING) young (E)

that would, presumably, scan the employee database to find an employee who is 25 or younger.

Another kind of client is a knowledge engineer who is interested in exploring or debugging some set of rules. RULOG provides a dialogue interface for this client, offering a set of commands that permit the user to stop the interpreter at various points in attempting a proof; to examine the values of variables, literals, and so on; to request explanations of why a particular predicate was determined to be true or why it gailed to be true; and so on. We refer the interested reader to the RULOG user manual (see [Cheatham]) for further discussion of the dialogue facilities.

4 The RULOG Interpreter

The RULOG interpreter has two major components – the reader and the prover. The reader accepts definitions, assertions, and dialogue comands, performs appropriate syntactic and semantic checks and creates an internal representation for items defined. Thus, the reader is analogous to the syntactic and semantic analysis components of an Ada compiler. The source of input is usually one or more files but input can be typed in directly as well. Syntactically invalid statements are rejected and an appropriate comment on the problem encountered is given; a statement can be re-submitted after editing if the user is typing directly. The semantic analysis resolves the types of overloaded literals and ensures that the type of each construct is consistent with that required; semantic errors also result in the rejection of the statement input accompanied by appropriate comments on the problems encountered.

The prover is invoked by being given some theorem to be established. In a fashion analogous to most logic program interpreters, it attempts to build a proof tree in order to establish that the theorem is true. The proof tree has as its root the predicate to be established. In general, at any point in the proof, the prover is working on some node of the proof tree in an attempt to establish that the predicate at that node is true. The operation of the prover at a given node is dependent upon the sort of predicate at the node; for discussion purposes we classify the predicates at a node into three groups, as follows:

Uninterpreted Predicates

An uninterpreted predicate is a predicate whose truth is established by appealing to the assertions that have been made about that predicate (that is, the rules whose conclusion involves the same predicate symbol as that of the predicate we are trying to establish). The processing of a node with an uninterpreted predicate is analogous to the processing done by a PROLOG interpreter. That is, to satisfy such a predicate we must find some rule whose conclusion has the same predicate symbol and such that we can unify each term of the predicate at the node with the corresponding

term of the rule conclusion. If successful, the premisses of the rule are established as the descendants of the current node and we turn to the next node of the proof tree; if not successful, the prover must backtrack and attempt to prove the predicates at some previous node in a different way (that is, a different rule or a different fact from the database).

Database Predicates

The truth of a database predicate is established by appeal to the appropriate relation in the relational database. At the present time we are using the TROLL database system but think that the modifications required to use some other database system would be minor. The transactions required between RULOG and the database system depend both on the predicate to be established and on the arguments to that predicate. We consider a couple of examples using the has_age predicate cited earlier. Suppose that we wanted to establish the truth of

has_age(e,a)

and that, at the time we wished to establish this, e was bound to "Henry" and a to 35. The transaction required is a query to the database that will determine and report whether there is a tuple of the employee relation whose name and age fields are 'Henry' and 35, respectively. If, after a subsequent failure to prove some predicate, we backtrack to this node there is no other way to prove it and backtracking will have to continue on to retry nodes previous to this one.

If, for the same example,

has_age(e,a)

both e and a were unbound quantified variables at the time we wished to establish that the predicate was true, a rather different transaction with the database would be required. This time we would request the values of the name and age attributes of the first tuple of the employee relation and bind the variables e and a to these values to establish the truth of the predicate. Upon backtracking to the node with this predicate, the transaction required is to retrieve the values of the name and age attributes for the next tuple of

the employee relation, continuing, during subsequent backtracking to the node, until the tuples were for the employee relation were exhausted before backtracking further.

Interpreted Predicates

Analogous to PROLOG, RULOG has a number of interpreted predicates including cut, fail and the like to control a proof. Like PROLOG, RULOG also provides the equality (=), disequality($\neq$) and inequality($\leq$) predicates but has a different interpretation of them that we will discuss below. RULOG also provides for assignment of new values to literals. An example is the "predicate"

$$S := \text{even}$$

that assigns the value even to the literal S and returns "true"; this ability to modify certain variables during a proof makes reasoning about situations that involve some notion of "state" rather more perspicuous than is often the case with PROLOG.

RULOG's treatment of $=$, $\neq$, and $\leq$ are different from PROLOG's. In PROLOG $x = y$ is true if x and y are manifestly equal or one of them is a variable that can be bound to the other; $x \neq y$ is the failure to show $x = y$; and, $x \leq y$ is valid only if both s and y are integers (including variables bound to integers and arithmetic operations on integers) with the obvious interpretation.

By contrast, the RULOG meaning of these predicates is provably equal, disequal, or inequal. We can think of the prover, when it encounters one of thse predicates, as appealing to a specialist who determines whether the given instance of the predicate is true or not and reports back accordingly. The specialists provided in RULOG are, as we noted earlier, based on the satisfiability procedures developed by Nelson (see [Nelson 81]) and incorporated in the Stanford Program Verifier. We term these specialists E and R. E maintains a conjunction of equality and disequality facats that have been asserted; the equalities are, in general, over terms constructed from tokens, literals, and uninterpreted functions applied to terms, recursively. E partitions the terms into equivalence classes and propagates each new

equality asserted so that, for example, the assertion x = y will cause x and y to be placed in the same equivalence class; E also propagates the equality so that, for example, f(x) and f(y) will be placed in the same equivalence class. Disequalities are managed by associating with each equivalence class the list of terms that are disequal to it; an attempt to add such a term to the equivalence class that it is forbidden to inhabit will result in unsatifiability. To prove that some equality or disequality is true, we demonstrate that conjoining its converse to E is unsatisfiable. E employes some fairly elaborate data structures and carefully chosen algorithms that achieve a time cost that is of the order of n log n to add an n-th conjuction to n-1 already known to E.

The R specialist maintains a conjunction of inequalities that have been asserted. It converts each inequality to an equality by introducing a so-called restricted variable; restricted variables are constrained to be non-negative and R insures that these constraints are met. The addition of a new inequality may result in the discovery of one or more equalities that are implied by the new inquality (as, for example, $z \leq x$ added to $x \leq y$ and $y \leq z$ would result in x=y and y=z being discovered); all equalities discovered are reported to E. The addition of a new inequality might also result in a restricted variable being negative, in which case the set of inequalities submitted to R is unsatisfiable. To prove that, for example, x>y, R shows that the addition of the converse, $x \leq y$, to the set of inequalities it currently has results in unsatisfiability.

The R specialist is also used to insure that the range contraints associated with types and subtypes are not violated. Suppose we have

```
type small is new INTEGER range 1..100
.
.
.
assert forall(..., V:small) p(...V...) if ...
```

Whenever the cited rule for p is involved in a proof, we must insure that $1 \leq V$ and $V \leq 100$. This is handled by submitting the two inequalities to R, and, if any subsequent assertions about V contradicts the constraint, R will report out the unsatisfiability; this is interpreted as a failure in the proof and initiates backtracking.

Future Directions

We think of the present RULOG as a prototype in which have demonstrated the feasibility of combining the basic mechansism of PROLOG with a type system, a set of specialists that are very efficient at determining the satisfiability of predicates over restricted domains, and a connection to a database system to provide a source of ground facts. In addition, RULOG provides a reasonable user interface and facilities for explaining why a proof succeeded or failed.

There are a number of additions to RULOG that we intend to investigate before we do the final round of engineering to insure that it is a reliable and robust system appropriate for distribution.

At present, RULOG has no mechanism for dealing with arbitrary collections of objects. We have rejected the idea of incorporating lists in the way that PROLOG does to provide for dealing with collections. Instead, we are exploring the possibility of adding sets to the language, complete with the usual set operations, set construct, set iterators, and the like. The experience with SETL (see [Schwartz 74]) and work by Sandhu see [Sandhu 81] suggest that the use of sets and set notations might be a very natural and user-friendly way to deal with arbitrary collections.

Another addition that is required for many applications is some notion of certainty (or, equivalently, fuzzy predicates); [Shapiro 83] disucsses how this might be added to PROLOG and we believe a similar addition to be possible to RULOG.

At present, the connection between RULOG and the database system (TROLL) is rather loose – RULOG runs on an Apollo and the database system on a VAX. We intend to investigate both connections to other databases and a tighter coupling of the RULOG and TROLL processes (possibly even combining them into a single process on one computer).

The versions of E and R currently operational in RULOG are satisfiability procedures, not decision procedures (that is, they do not bind quantified variables). We believe it straightforward to make E into a decision procedure and are exploring various ways to extend R to be able to do variable binding as suggested in [Townley 80].

REFERENCES

[Cheatham] Cheatham, T. E., The RULOG User Manual. In preparation.

[Feigenbaum 83] Feigenbaum, E. A., and McCorduck, P. *The Fifth Generation.* Addison-Wesley, 1983.

[Mishra] Mishra, P., Polymorphic type inference in PROLOG. Extended summary. CS Department. University of Utah, Salt Lake City, Utah.

[Nelson 79] Nelson, G., and Oppen, D., Simplification by Cooperating Decision Procedures. *ACM Trans. on Programming Languages and Systems, 1,* 2 (October 1979), 245-257.

[Nelson 81] Nelson, G., Techniques for Program Verification. CSL-81-10, Xerox Palo Alto Research Center, June, 1981.

[Sandhu 81] Sandhu, R. S., The Case for a SETL Based Query Language, LCSR TR-24, Rutgers University, 1981.

[Schwartz 74] Schwartz, J. T., On Programming: An interim report on the SETL Project. Installments I and II. CIMS, New York University, 1974.

[Shapiro 83] Shapiro, E., Logic Programs with Uncertainties - a Tool for Implementing Rule Based Systems, *IJCIA 1,* (1983), 529-532.

[Townley 80] Townley, J. A., A Pragmatic Approach to Resolution-based Theorem Proving. *Int. Jr. on Computer and Information Sciences 9,* 2, (1980), 93-116.

END

FILMED

4-85

DTIC